A MIDDLEWARE INDEPENDENT SERVICE MODELING AND A MODEL TRANSFORMATION FRAMEWORK

BY

WEERASAK WITTHAWASKUL

B.E., King Mongkut's Institute of Technology, 1991
M.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

A MIDDLEWARE INDEPENDENT SERVICE MODELING AND A MODEL TRANSFORMATION FRAMEWORK

Weerasak Witthawaskul, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2006
Ralph Johnson, Advisor

One of the reasons enterprise software is difficult to change is because it relies on middleware services. Middleware makes the rest of the application simpler but then the application depends on the middleware. Migrating existing software to new middleware technology requires significant software redesign, rewriting and testing.

This dissertation follows the Model Driven Architecture (MDA) initiative from the Object Management Group (OMG). The MDA separates business or application logic from underlying platform technology by defining application models at two levels; a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The dissertation focuses on the middleware independent aspect of the platform. By specifying common services using UML profiles that do not depend on specific middleware technology and providing separate transformations for each service, it is possible to develop business software models at a middleware independent level and to use transformers to map a PIM onto a middleware specific implementation.

The Mercator model transformation tool and framework help two kinds of developers. This first kind is modelers who use the tool to edit PIMs and use common object services packaged as profiles. The second kind is model compiler developers who define profiles and create transformers that translate the PIMs into middleware specific implementation. The framework provides the profile definition and transformation structure as well as model manipulation APIs that allow model compiler developers to plug in new object services and/or new transformations of existing services. PIMs no longer contain

middleware specific information. This information is customizable and kept separately in annotation files.

Our contributions in this research are:

- A systematic method to define middleware independent object services.
- Profiles for object persistence, naming, distribution, transaction and messaging services. These profiles contain stereotypes, APIs, annotations and model transformers.
- A novel, lightweight, stereotype-triggered model transformation framework that allows object service transformers to plug-in to manipulate model elements.
- A modeling language based on eMOF with support for UML profiles and viewpoints.
- A Mercator tool that implements the framework.

We believe that this research is a step towards the realization of the MDA approach.

To Mom, Dad, Sister and Brother

Acknowledgements

This dissertation would not have been possible without the support of many people. My first acknowledgement is to my advisor, Ralph Johnson. He is a great teacher who diligently teaches not only technical aspects but also ways of life, a good listener who understands what I sometimes missed to convey, and a brilliant mentor who has patiently given me practical advice and encouragements during my entire study. I am grateful to know and work with him. I am also thankful to my committee members, Mehdi Harandi, Samuel Kamin and Jennifer Hou, who gave suggestions and insights to improve the dissertation.

My next thanks are given to Professor Adrienne Perlman from the department of speech and hearing science for whom I have worked in the telemedicine project for many years. She has given me valuable advice, financial support, and a project opportunity for developing a software system that can improve healthcare to patients with swallowing disorder. It has been a great experience and fun to work with colleagues from the Dysphagia research lab and those from the University of Arkansas, the University of Wisconsin at Madison, Walter Reed Army Medical Center and Landstuhl Regional Medical Center. I am also grateful for financial support from the NCSA and Caterpillar during my database editor project in my first two years of study.

I am thankful to the software architecture group whose members have helped giving constructive feedbacks for my papers and presentations, especially Paul Adamczyk who read and commented every chapter of this dissertation. They are the same group who helped me organize the pattern language of programs (PLoP) conference when I was the conference chair in 2002. I am also thankful for anonymous reviewers who gave numerous constructive feedbacks in my papers.

Lastly but most importantly, I would like to thank my family and numerous friends I have met here. I am grateful for my parents who raised me to become who I am now. They showed me the importance of education and gave me a freedom to pursue my dream. I

also greatly appreciate my brother and sister who have taken care of them while I am far away from home.

Weerasak Witthawaskul
Urbana, Illinois

Table of Contents

List of Figures

List of Tables

List of Code Listing

Chapter 1 Introduction

1.1 Overview

"Enterprise computing" is another name for business applications in large companies. Business applications are usually data intensive; access data from dispersed locations; interact with business users from all departments inside the companies as well as customers, investors and suppliers. As companies expand their businesses, merge with others that use different application packages and incompatible infrastructure, system integration becomes important and complicated. Most modern enterprise applications are implemented in object oriented languages like Java or C# and use a software layer that provides common object services such as persistence, distribution, transaction, messaging, among others. This software layer is known as middleware. There are many middleware products in the market. Some such as CORBA, J2EE and .NET provide many object services while others provide specific object services. Middleware like CORBA can put an object face on software written in non-object-oriented languages since a lot of existing software is not at all object-oriented. Enterprise computing must be flexible to support legacy, current and emerging technology.

Each middleware package tries to solve a standard problem of enterprise computing. However, they also add to the problem, because each middleware has different application programming interfaces; thus applications that use these middleware depend on them and it is difficult and often expensive to restructure existing programs to switch from one middleware to another. Table 1 - 1 shows many different middleware libraries that can be used for each common object service.

Common Object Services	*Middleware Libraries*
Persistence	EJB CMP, Hibernate, JDO, JDBC/SQL, XML, Toplink, SDO
Transaction	EJB CMT, database monitor, JTA
Naming	JNDI, Web Service UDDI, URN, UUID
Distribution	RMI, CORBA IIOP, SOAP/XML
Messaging	JMS, ESB, Message Driven Beans, Axis Web Service, MQSeries

Table 1 - 1 Choices in Middleware Products

As technology evolves, it is inevitable that there will be more and better middleware. Companies need to find a better software development approach that abstracts middleware concerns from their software models. This dissertation defines a middleware independent common object services as profiles that can be used in middleware independent models, and provides an object oriented framework that translates software models that use these profiles into executable implementations. Software developers create models that use common object services provided by profiles and choose concrete middleware libraries that implement those services. Software models no longer contain concrete middleware specific information and therefore changes in middleware libraries will not change the structure of the software models. This will enable enterprise applications to be modeled independently of particular middleware and then translated to a form that uses the middleware. The next section shows an example of a small program that must be implemented differently if it uses different middleware and shows that middleware independent modeling reduces complexity in the software design.

1.2 A Motivating Example

This example shows a scenario where middleware increases complexity in a software application. Suppose we create a simple project for a financial company. We start from a high level design of the application with a small set of requirements. Then we will develop the application into an implementation using Java RMI. Later on, when another

business case requires the implementation to move to CORBA, the same high level design will be re-implemented. We want to show that it is more effective to generate different kinds of implementations automatically from the same platform independent model than it is to manually re-implement the system again and again.

Suppose a company wants to create a stock market application to obtain the current stock prices of its clients. The application consists of a stock market object and a client. A client can obtain the stock price of any stock symbol. The stock market object contains a public method getPrice() and returns the current stock price of an input stock symbol. The client creates a stock market object and invokes a getPrice() method to obtain a price for a input stock symbol and prints the price on an output console. Figure 1 - 1 below is the UML class diagram of the stock market domain.

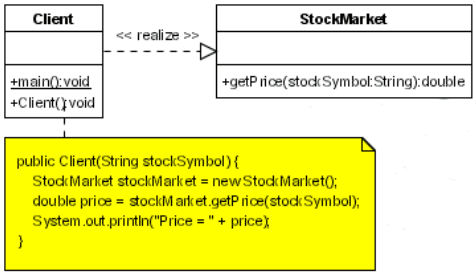


Figure 1 - 1 Conceptual class diagram

Figure 1 - 2 Both client and stock market objects are in the same deployment node

If both the client and the stock market are in the same node (Figure 1 - 2), one possible implementation is:

```
public class Client {
  public Client(String stockSymbol) {
    StockMarket stockMarket = new StockMarket();
    double price = stockMarket.getPrice(stockSymbol);
    System.out.println("Price = " + price);
  }
  public static void main(String[] args) {
    new Client(args[0]);
  }
}

public class StockMarket {
  public StockMarket() {
  }
  public double getPrice(String stockSymbol) {
    // Read data from the database
    return ...;
  }
}
```

Listing 1 - 1 Client.java and StockMarket.java

This model is fine for conceptual modeling. Business modelers do not care how the model is implemented or where each object is executed as long as the client can invoke the getPrice() method from the stock market object. However, software developers have to consider the platform and system configuration before they can create an implementation. In this case, there will be many client objects, probably as many as a few hundred, and most reside in a different operating environment than the stock market object and therefore cannot invoke the method of the stock market directly (Figure 1 - 3). So we need to be able to invoke a method from a remote object. This remote invocation mechanism is specific to a middleware or programming technique and we need to make a decision on which platform we will choose to implement this requirement. Notice that the server object creation is now moved from the client to the server node.

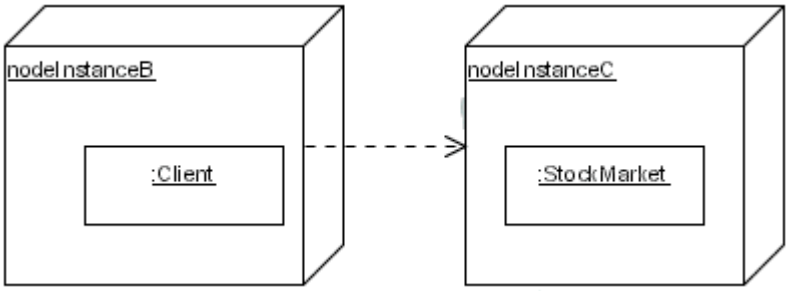


Figure 1 - 3 A remote invocation is required when both objects are running in different nodes.

If the company uses Java as its primary programming language then it is logical to use Java Remote Method Invocation (RMI) to achieve this. We refine the PIM in using the following steps. The result model is shown in Figure 1 - 4.

1. Create a remote interface, StockMarket. This interface extends the java.rmi.Remote interface. Define the getPrice() method that throws a java.rmi.RemoteException.
2. Create a class, StockMarketImpl, that implements the StockMarket interface. Create a constructor that takes a name and can throw a java.rmi.RemoteException. In the class constructor, bind itself to a java.rmi.Naming object. Implement the getPrice() method.
3. Create a server object, StockMarketServer. This server creates an RMI security manager and initializes the stock market object with a name. This name will be used for the binding of the rmiregistry.
4. At the client, an RMI security manager is created and the remote stock market is looked up by the java.rmi.Naming using the name defined in step 3.
5. The client can invokes the getPrice() method and obtain the result.

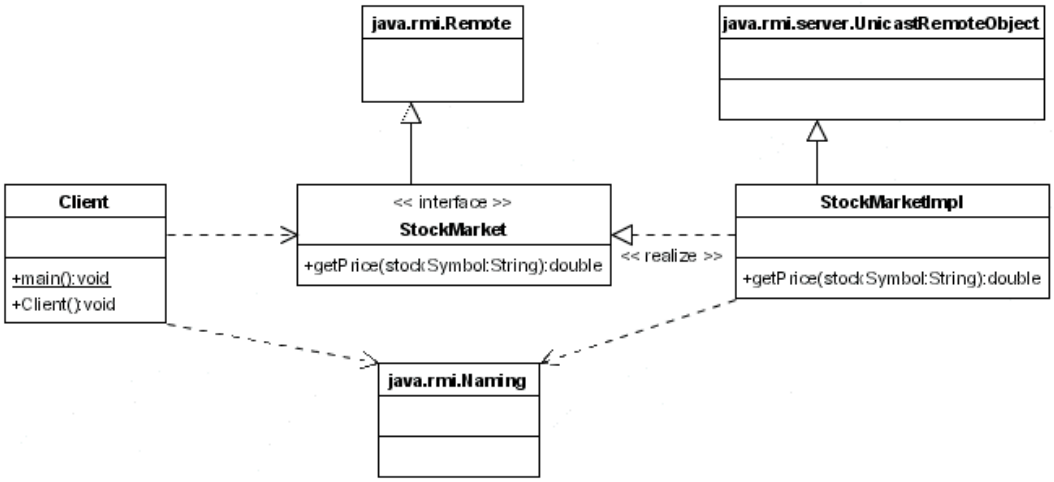


Figure 1 - 4 shows the class diagram of the stock market model using Java RMI.

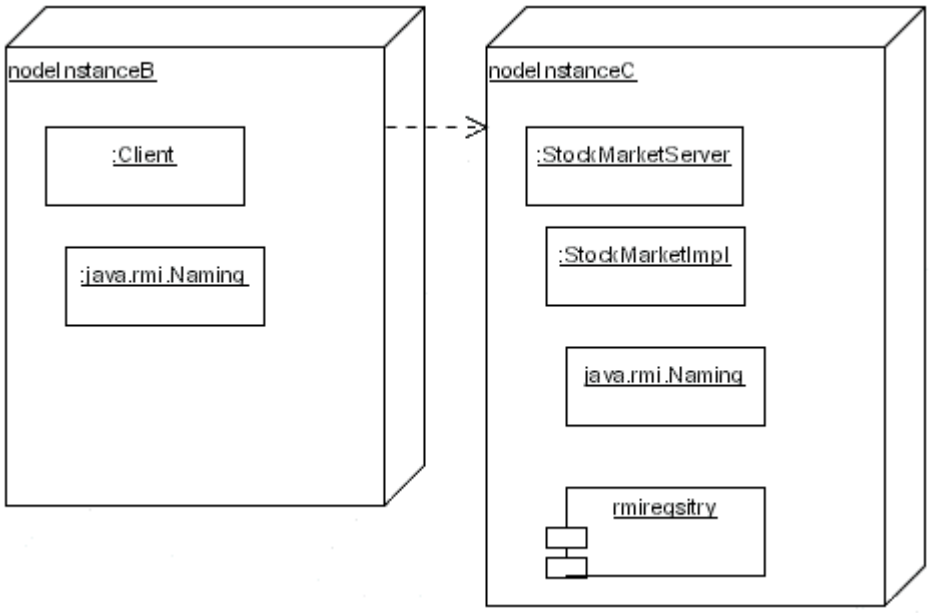


Figure 1 - 5 A deployment diagram for the Java RMI implementation.

Before we execute the application, we need to start a Java naming and lookup service program called *rmiregistry*. Once the program is started, we execute the server application, *StockMarketServer* which will instantiate the *StockMarketImpl* object. The object binds itself to a name supplied by the server application. At the client side, we execute the *Client* which must know where the *rmiregistry* is and the name of the *StockMarketImpl* object. The deployment diagram is depicted in Figure 1 - 5. Listing 2 – 4 show the source code for the Client.java, StockMarket.java, StockMarketImpl.java and

StockMarketServer.java respectively. Notice that the *StockMarket* which is a class in our initial design now becomes a Java interface and its implementation is by the *StockMarketImpl* class.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class Client {
    public static void main(String[] args) throws Exception {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        StockMarket stockMarket =
            (StockMarket) Naming.lookup("rmi://192.168.0.2/NASDAQ");
        System.out.println(
            "The price of APL is " + stockMarket.getPrice("APL"));
    }
}
```

Listing 1 - 2 Client.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface StockMarket extends Remote {
   double getPrice( String symbol ) throws RemoteException;
}
```

Listing 1 - 3 StockMarket.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class StockMarketImpl
    extends UnicastRemoteObject
    implements StockMarket {

    public double getPrice(String symbol) {
        return 55.0d;
    }

    public StockMarketImpl(String name) throws RemoteException {
```

```
        try {
            Naming.rebind(name, this);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Listing 1 - 4 StockMarketImpl.java

```
import java.rmi.RMISecurityManager;

public class StockMarketServer {

    public static void main(String[] args) throws Exception {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        StockMarketImpl stockMarketImpl = new StockMarketImpl("NASDAQ");
    }
}
```

Listing 1 - 5 StockMarketServer.java

The system built on Java RMI worked fine. One day, after the company installed a firewall which disabled all socket communications, the development team decided to slightly change the implementation to use RMI over IIOP. It was quickly done but the distribution of the new software version to all clients took some time. Later on, customers of the company wanted to connect to this system from their CORBA-based applications. The company then decided to migrate their system from Java RMI/IIOP to CORBA using the default ORB support in Java2. The development team went back to the conceptual diagram in Figure 1 - 1 and reimplemented the application using the following steps:

1. Define a CORBA IDL for the stock market interface. Name it stock.idl.

```
module Stock
{
   interface StockMarket
   {
```

```
        double getPrice(in string stockSymbol );
    };
};
```

Listing 1 - 6 stock.idl

2. Use Java's idlj.exe to compile the stock.idl into stub (_StockMarketStub.java) and skeleton (_StockMarketImplBase.java) as well as helper files (StockMarket.java, StockMarketHelper.java, StockMarketHolder.java, StockMarketOperations.java).
3. Implement StockMaket interface in StockMarketImpl.java
4. Create the StockMarketServer class. Initialize the CORBA Orb, create the stock market object and connect it to the Orb. Write the stock market object string into an IOR file.
5. Write the Client class. This class takes an IOR file as an input, creates a CORBA's object from the input string and narrows the object into the StockMarket interface. Invoke the stock market method and display the result.

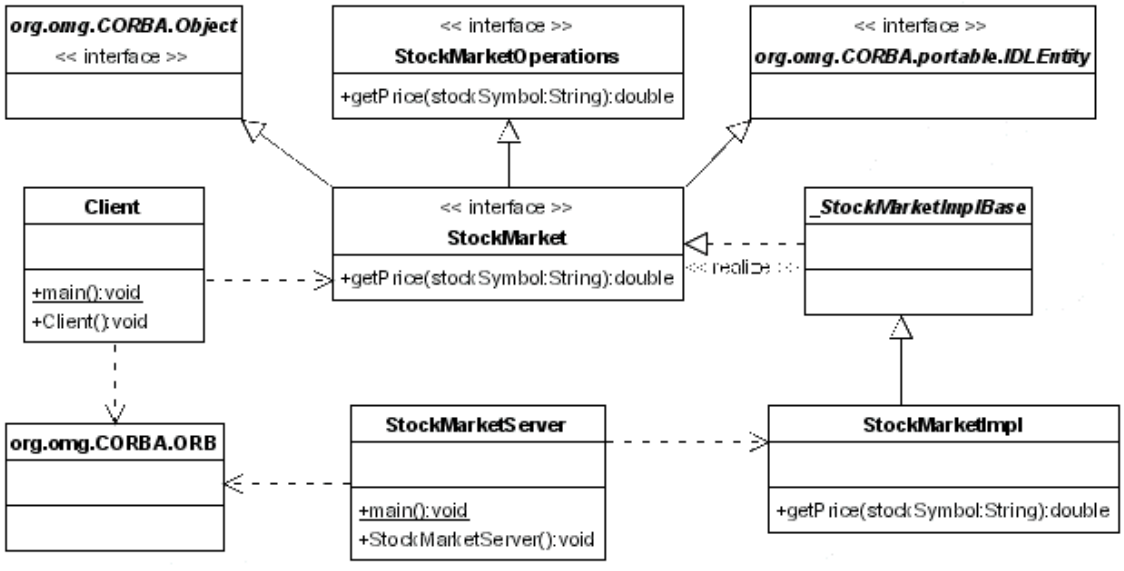


Figure 1 - 6 The class diagram of the same stock market model using CORBA

To execute the CORBA implementation, we run the StockMarketServer on the first computer. The server will generate the IOR file that will be used as an input to the Client program. The client obtains an object reference and invokes the remote object operation via the stub at the client and the skeleton at the server.

If the company moves to a commercial ORB product that supports CORBA's basic object adapter (BOA) or portable object adapter (POA), it needs to change the model and the implementation again. What if the company adopts web service and uses SOAP? Do they have to re-implement the same problem over and over again?

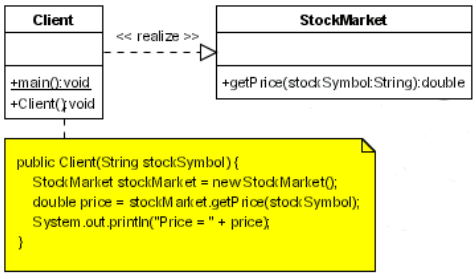


Figure 1 - 7 The stock market class diagram

We notice that changes in platform require reimplementation. Different flavors of Java RMI and CORBA implementations do not require changes to the conceptual model we created in Figure 1 - 7. However, the implementation steps are different and it's likely that the existing implementation is discarded and a new implementation is created from scratch. If the company does not maintain the conceptual model in Figure 1 - 7, a change from one implementation, i.e., Java RMI, to another, i.e. CORBA, will require extensive reengineering of the existing implementation. Since models using a particular technology usually contain platform specific idioms, extra classes and naming conventions, this makes it difficult to analyze a platform specific model and to convert it to use another technology. As shown in the above example, it's easier to convert the model in Figure 1 - 7 to Java RMI or CORBA implementations than to convert a Java RMI to a CORBA implementation or vice versa.

Figure 1 - 7 is an example of a platform independent model (PIM) while Figure 1 - 4 and Figure 1 - 6 are examples of platform specific models (PSM). We want to show that it's possible to automatically transform a PIM to different PSMs so that developing an application would involve only one PIM as shown in Figure 1 - 8. The extra information required for platform specific transformation such as object partitioning or communication protocol should be kept outside the PIM.

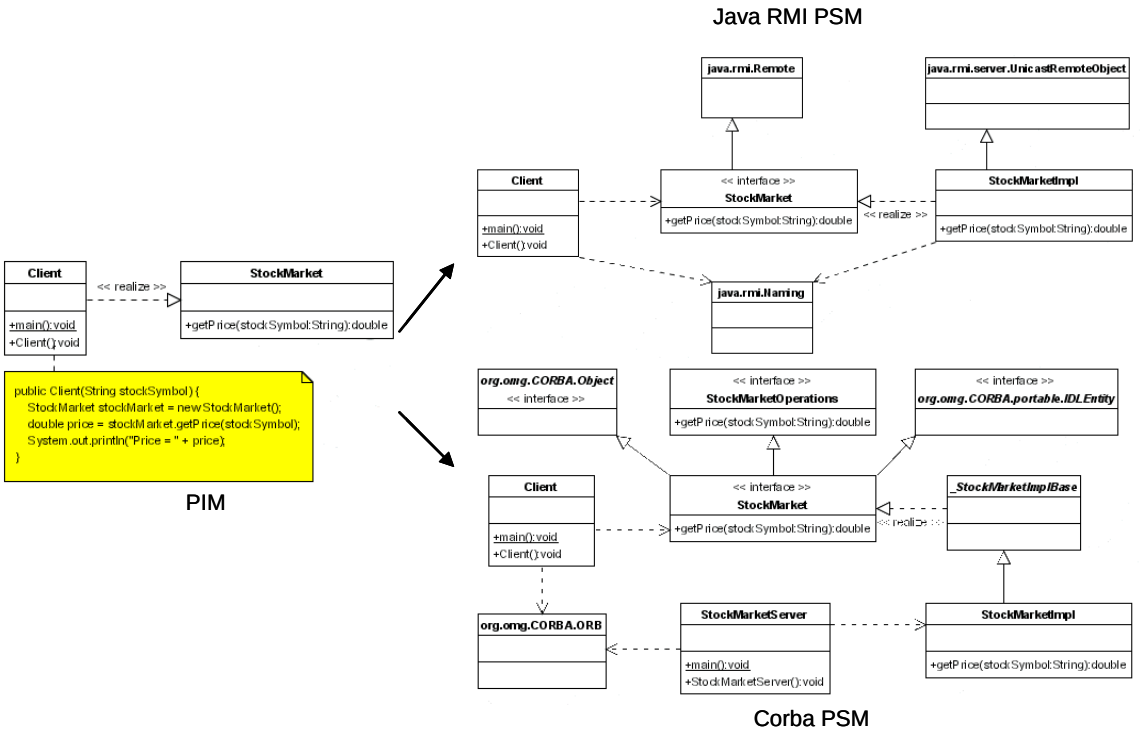


Figure 1 - 8 The PIM is implemented with different technology

This example illustrates only a small problem in object creation and invocation. We still don't answer questions such as:

1. How do we use a naming service independently of any specific technology so that we can use it in any PIM? Moreover, how can we use other common services that are not specific to any middleware or platform?
2. What should be stored in a PIM and what should be stored in an annotation model? How can we know that the PIM is complete and ready for PIM-PSM transformation?
3. How should we design a transformation tool that is flexible enough to transform a PIM to current PSMs and support future PSMs that use different technology, platform, and middleware?

1.3 Middleware Independent Libraries

David Frankel emphasizes the importance of middleware independent modeling in his book, Model Driven Architecture: Applying MDA to Enterprise Computing [Fra03].

> *"The value of a middleware-independent model is that, together with generators, it raises the abstraction level even above middleware. Now that multiple middleware technologies have proliferated, developing directly to specific middleware carries platform volatility risks. In many cases, therefore, it is preferable to define a middleware-independent model that can be mapped to particular middleware. As we've seen, raising the abstraction level tends to improve all of the viability variables."*

In a distributed computing system, middleware is defined as the software layer that lies between the operating system and the applications on each site of the system [Obj05]. Companies use services provided by middleware to manage persistence data, reliable messaging, transactions and object naming. However, middleware is another piece of software that evolves. New middleware technology emerges and is often incompatible with existing ones. Companies have to maintain legacy systems as well as new ones. Changes in middleware adoption are costly and often avoided. For example, an e-commerce system that migrates its object persistence technology from one technology to another from Enterprise JavaBeans [Sun06] to Hibernate [Hib06], requires substantial changes in source code and deployment configurations. It's difficult to understand the domain model and extract the actual business logic inside the model because the domain logic is often hidden among middleware specific details. A PIM should not contain these details. Instead, common services such as object persistence, distribution, transaction, messaging and security should be well defined and used by importing from middleware independent libraries. To specify object persistence, for example, modelers should be able to import the object persistence library, specify which classes whose instances need to be persisted, which class attributes are object identities and the APIs to store, load and find objects from persistence storage. These libraries will be mapped by library transformers into specific implementations that are selected during the model transformation. For example, an object persistence library may contain transformers for

an EJB Container Managed Persistence (CMP) transformer and Hibernate among others. Which transformer is used depends on the mapping method selection.

Once common object functionalities have been separated into libraries, it should be simple to provide new mappings for existing libraries and to define new libraries. Model developers from different industries may want to use standard business models and tailor them to fit their domain specific needs. They may want to implement common object services differently. To do so, they develop their own transformers and describe how and where these transformers are executed.

Middleware independent libraries are important in two ways. First, model designers use them in PIMs to solve their business problems without knowing details about how their models and libraries are mapped into the target platform. Modelers decide on mapping choices and provide extra information during the PIM-to-PSM transformation. These choices and extra information are stored in an annotation model thus separating PIM from implementation details and making the same PIM transformable into different platforms. Second, model compiler developers use platform independent libraries as contracts to implement new mappings that are more specialized or support new target technology.

However, providing middleware independent libraries is difficult. Many libraries depend on one another. For example, object persistence requires object identity to provide object key while object distribution uses object serialization to transport objects. Sometime implementation choices may conflict with each other. For example, object distribution cannot transport objects that are stored in a relational database. One implementation choice may constraint another, for example, Hibernate always stores objects in relational table data source. Therefore choosing a file data source with Hibernate is not valid. This makes it hard to design libraries that are reusable and support dependency management.

Another potential problem in using high level, middleware independent APIs is that it may take a lowest common denominator approach and not be as comprehensive as

platform specific APIs. It is also hard to provide APIs that can be mapped into many different implementations. However, this is always a case for a higher level of abstraction such as a virtual machine. As long as the virtual machine is ported into a different operating environment, programs that run on the virtual machine can use virtual machine APIs without knowledge of the actual runtime environment. If the platform independent APIs are well defined, it is a matter of providing the mappings of these APIs into a new runtime platform and reusing the same PIM to generate different implementations running on different technology, middleware or platform. The mapping of platform independent models and libraries is an important part of a model transformation.

Designing stable platform independent APIs is a difficult problem. Library designers want to expose platform independent APIs at a level of abstraction that PIM can use and at the same time, contains enough information so that the PIM can be transformed into different concrete implementations. We propose middleware independent APIs as a set of interfaces grouped by application-level, commonly used services that can be reused over and over. We show that we can implement these APIs with middleware specific code. Therefore, PIM APIs become black box interfaces in each middleware service. The task of defining specification consists of finding middleware service interfaces and extending modeling elements to support those interfaces in terms of UML stereotypes, stereotype attributes and constraints. Chapter 3 to 7 define these interfaces for different middleware libraries.

1.4 Model Transformation

Model transformation is the process of converting a model described in one modeling language into another model described by a potentially different modeling language. A model transformer is similar to a compiler. A compiler parses textual source files and builds an abstract syntax tree (AST) representation. It then traverses the AST and generates intermediate representations potentially in several passes. Each pass contains more concrete intermediate representation and optimization. The final result is an executable under a target machine. Similarly, a model transformer parses an input PIM

from a textual representation format in XMI and builds a model tree. The transformer traverses the model tree and generates intermediate, refined models potentially in several passes. Each pass contains extra information from mapping algorithms or user parameters. The final result is a PSM that can directly be transformed into source files. There are many ways to transform models as described in section 1.5.4. There are many ways to transform models. Czarnecki described a classification of different model transformation approach [Cza03]. Our approach uses an object oriented model transformation framework. Our framework, Mercator, allows model compiler developers to create object service transformers and plug them into the framework to support new object services and/or add middleware specific transformations to the existing object services. The framework is described in chapter 2.

1.5 Related Work

There are two research spaces related to this dissertation. One space is a methodology and specification research. The OMG's Model Driven Architecture is a fundamental inspiration to this dissertation. Microsoft proposes a domain specific language approach in their Software Factories. The Enterprise Distributed Object Computing group proposed a set of profiles that addressed several object services. The other space is a model transformation research. There are many transformation approaches based on graph and relation theories, declarative and generative techniques. The OMG's QVT is one approach for declarative model-to-model transformation. The IBM Model Transformation Framework is a transformation mechanism based on relations. The Parallax framework is a closely related research that attempts to address the specification and transformation of middleware services.

1.5.1 Model Driven Architecture

The MDA is an initiative of the OMG to allow enterprise applications to be platform independent [MDA01]. Our approach follows the MDA to raise the abstraction level of software. It is similar to the MDA in that it separates business or application logic from underlying platform technology by defining application models at two levels; a Platform Independent Model (PIM) and a Platform Specific Model (PSM). The PIM is a high

level model that does not depend on middleware, platforms, operating systems, programming languages and technology whereas the PSM is a complete, executable model for a particular target platform. This approach focuses on models which are representations of things and usually described by modeling languages. Models can be described graphically or textually. The OMG proposed the UML [UML05] as a common modeling language for representing the PIM and the PSM but it is not the only one. For example, a model can be a Java program. Its modeling language is essentially the Java programming language. The OMG defines a modeling framework, Meta Object Facility (MOF) as a standard way to describe modeling languages. Therefore, any object oriented programming languages that can be described by the MOF are also modeling languages. Therefore, the MDA is applicable to visual models such as UML as well as textual models from programming languages. The same model can be represented visually as diagrams or textually as program fragments; depending on "viewpoints". Working with model viewpoints can thus reduce the complexity of the models by hiding information that is irrelevant to the current work at hand. It is useful when the models are large. A joint effort between ISO and ITU-T resulted in a Reference Model for Open Distributed Processing (RM-ODP) that defines 5 viewpoints; namely enterprise, information, computational, engineering, and technology [Ray95]. This dissertation focuses on the technology aspect and the middleware independence in particular.

Designing software at a platform independent level focuses on the problem domain and leaves out specific implementation details. Software developers design an object model in a PIM; apply ("mark") common library concepts such as persistence, transaction, distribution, messaging in the PIM and refine ("annotate") the marked PIM with specific implementation details for each PSM (Figure 1 - 9). A model mapping tool generates a PSM from the marked PIM and the platform specific annotation. The PSM can be executed in a modeling environment using a virtual machine approach [RFBO01] or translated into executable artifacts (code and configuration files) using a code generation approach [CE00]. The MDA reuses a PIM because the same PIM can be used to generate different PSMs by using different annotations and mapping rules.

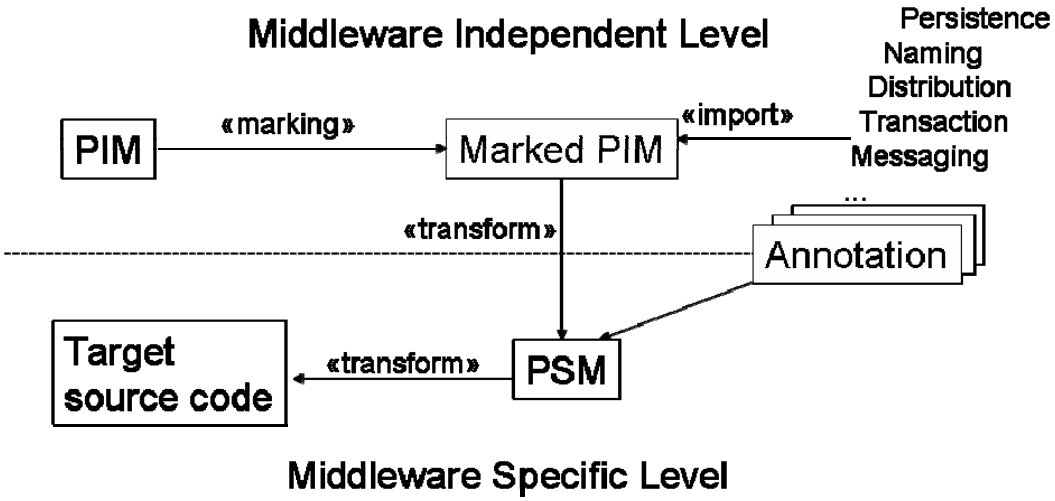


Figure 1 - 9 PIM, PSM and mapping process

MDA has two kinds of developers. The obvious kind is model designers who are responsible for designing and developing models that solve particular problems. They design solution models, import and apply platform independent model libraries and do not worry about how the platform independent models (PIMs) are mapped into the platform specific models (PSMs) until at a later stage. This group is considered 'end users' to the model transformation. The other kind is model compiler developers who are responsible for defining platform independent libraries and providing the mappings between PIM and PSM. The separation of these two groups is similar to other software construction discipline such as the software product line [CN03] which focuses on the vertical construction of software in the same family. Product line developers create a framework for product families that can be customized by product developers in the same way as products are assembled from product lines in the manufacturing industry.

Our approach is dissimilar to MDA in that while MDA provides a high level framework and supporting standards for modeling, stereotyping and transformation, it does not describe how the standards are used and transformed. Our approach defines a concrete framework that manipulates models conformed to modeling languages, gives semantics to stereotypes grouped into profiles and provides plugins for programmable transformers. We believe that our approach is one implementation of the MDA concept without using all MDA standards.

1.5.2 Software Factories

Software Factories is the Microsoft alternative to MDA. It combines model driven development, software product lines, domain specific languages instead of UML to automate the construction of product families and reuse software assets in specific domains [GSCK04]. They achieve economy of scale by defining common production assets for a software product family in a given domain and developing family members using those assets. Developers build software products by assembling common software assets and customizing them. Product family developers define software factory schema that consists of DSLs, tools, assets and mappings for a product family. Instead of using UML, Software Factories use domain specific languages and XML as source artifacts. This is similar to our approach, we use a variation of OMG MOF instead of UML to store models and use XML to describe metadata. Software factory schema allows for a construction of different modeling languages. However, we differ in that we target the automation and reuse in middleware libraries instead of product family in each domain. Software Factories use pattern templates to generate target implementations while our approach uses imperative transformation in Java.

1.5.3 Enterprise Distributed Object Computing

The middleware independent library concept in our approach is similar to the standards from the Enterprise Distributed Object Computing (EDOC) group. The EDOC supports a distributed enterprise computing modeling by proposing enterprise collaboration architecture that uses component collaboration models at different levels of granularity to describe structure and behaviors of enterprise systems. The EDOC defines 3 platform independent services; entity, event notification and business process. According to the ECA, these services are described as models.

- Entities model describes a metamodel that may be used to model entity objects that are representations of concepts in the application problem domain and define them as composable components.
- Events model describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model event driven systems

- Business process model specializes the component collaboration architecture (CCA) and describes a set of model elements that may be used on their own, or in combination with the other EDOC elements, to model system behavior in the context of the business it supports.

Figure 1 - 10 below shows the UML profile of entities. The «Entity» stereotype labels persistence objects. The «Entity» accesses to the «Key» element from the «Entity Data» stereotype. We are not sure why there is no dependency between the «Entity» and the «Entity Data» but the description in the UML profile for the ECA [ECA04] indicates so. The «Key» element can be from a class attribute labeled by «Key Attribute» or a « Foreign Key» linked to another entity. Entities can play roles and can be nested into a composite. The «DataProbe» observes the «Entity» state and notifies other objects based on the kind of probe extents.

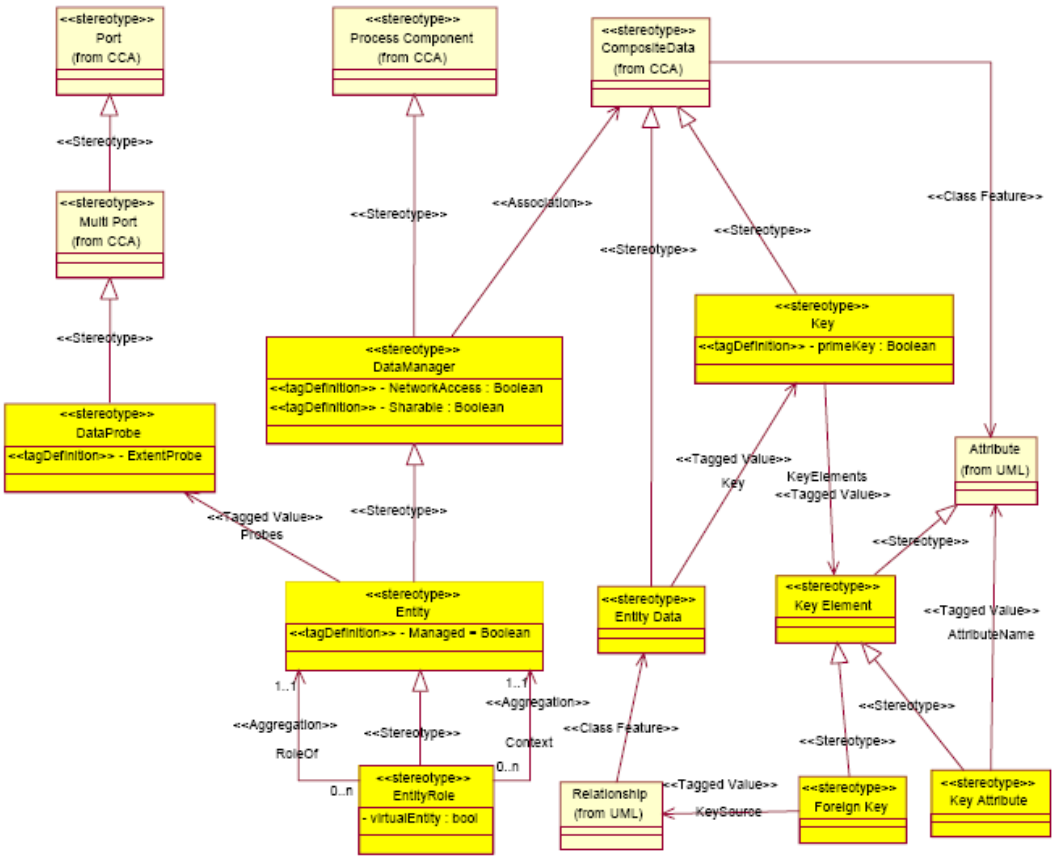


Figure 1 - 10 EDOC Entity Metamodel

There are differences between our approach and the EDOC specification. While the EDOC provides a coherent set of profiles that supports basic object services, it does not show how models that use these profiles are transformed into implementations. It does not cover other important services such as distribution and messaging that are commonly used in business applications. There are some detailed design questions in each EDOC profile. For example, the EDOC Entity Profile still does not address important persistence features. First, it does not provide persistence APIs and thus makes it impossible to programmatically store objects. Second, it does not provide ways to specify where object are stored. Can it store objects only to relational tables as the terms, key and foreign keys are borrowed from a relational paradigm? Third, it lacks a notion of a persistence manager. It is not clear how entity objects are atomically stored and retrieved from different data stores.

At the PSM level, the EDOC specifies a couple of UML profiles for Java and EJB [UmlEjb01] and CORBA middleware [UmlCor02]. However, the EJB profile is based on a very old version 1.1. The Java profile is not complete; it is modeled for basic elements (package, class, field, methods, and parameters). It can not model Java expressions inside method blocks. Nor does it support exceptions. Most importantly, the EDOC does not provide a reference implementation or describe how to map EDOC models into PSMs. Our approach provides a concrete implementation and describes a process of modeling, stereotyping and transforming models from a platform independent level to a platform specific level. We use factory objects to create and manipulate models from different modeling languages so it is possible to generate code that conforms to various versions of the same programming language like Java 1.1 and Java 5.0 by delegating the model manipulation to different factory objects.

1.5.4 Model Transformation

Model transformation is an active research area. The area consists of two main categories; model formalism and model transformation languages. There are several attempts to define model formalism [Cap02]. Kovse showed why it is important to have a generic model-to-model transformation [Kov02]. Czarnecki studied the classification

of model transformation approaches [CH03]. Most techniques are categorized in three groups. First group uses generative programming [Sil03]; the second is a template based, domain specific model transformation languages [Xsl99] [QVT05] [Mig02] [Tra05]; and the last group is a graph-based transformation [Agr02] [Sen03]. Sendall discussed the effectiveness of combining generative and graph-based transformation [Sen03]. Examples of model transformation are mostly based on structural mapping such as between data models [Gog02], between component models [Zia02], in specific domains such as realtime and embedded component middleware [Sch02] [GTT02], and enterprise applications [KVR02]. However, they do not take into consideration the behavior of the model at the operational level and do not make it possible to use services across domains. Gardner et al provided a detailed analysis of transformation frameworks submitted to the OMG QVT proposal and found out that each approach had strengths and weaknesses and recommended a hybrid approach [GGKH03]. Our approach uses a stereotype-triggered, object oriented framework and a generative technique to translate models in one modeling language to ones in another language. The framework provides a uniformed way for both model-to-model and model-to-code transformations.

1.5.5 MOF Query/Views/Transformations

The MOF Query / Views / Transformations (QVT) specification from OMG is of a particular interest from the model transformation community. The QVT is similar to our transformation framework in that it transforms an input model based on one modeling language into an output model based on another language. The QVT is currently in a final adopted specification status from the OMG. It supports model transformation based of MOF 2.0 and defines three domain specific languages; Relations, Core and Operational Mappings. At first, the QVT RFP tried to use declarative languages in all three languages to describe model relations and mappings between relations but it found out that the Operational Mappings are easily described using an imperative language. In contrast, we don't invent a declarative language. Instead, we use Java for model transformation. Another important dissimilarity is that the QVT does not support model-to-code transformation which is an important piece to generate an actual source code implementation. Our transformation framework uses stereotyping for both model-to-

model and model-to-code transformation. Thus it provides end-to-end model transformation from an input PIM to an executable code.

The QVT extends the Object Constraint Language (OCL) 2.0 to define model query and manipulation. The code below compares the code that returns all top level classifiers using QVT and Java.

```
-- QVT
query getBaseClasses(pack: ecore::EPackage): Bag(ecore::EClass) {
      pack.eClassifiers.oclAsType(ecore::EClass)->select
          (c | c.eSuperTypes->isEmpty())
}

// Java
public static EClass[] getBaseClasses(EPackage pack) {
    List baseClasses = new ArrayList();
    for (Iterator it = pack.getEClassifiers().iterator(); it.hasNext(); ) {
        EClassifier classifier = (EClassifier) it.next();
        if (classifier instanceof EClass == false) {
            continue;
        }

        EClass klass = (EClass) classifier;
        if (klass.getESuperTypes().isEmpty()) {
            baseClasses.add(klass);
        }
    }

    return (EClass[]) baseClasses.toArray(new EClass[baseClasses.size()]);
}
```

Listing 1 - 7 Model transformation comparison between QVT and Java

The QVT is a powerful declarative language. It is concise and integrated well with the MOF metamodel. It is also a standard by OMG that tool companies develop and support. However it is more complicated to learn and it is questionable how well the language scales to support complex transformations since the OCL was designed as a model query and checking language rather than the model manipulation language.

1.5.6 IBM Model Transformation Framework

The IBM Model Transformation Framework (MTF) is another DSL that use relation concept. However, the MTF use models based on Eclipse Modeling Framework (EMF) which is a variation of the essential MOF (eMOF) 2.0. It is a declarative approach that defines relations between two models and generates reconciliation (deltas) that makes two models consistent. A high level example of defining a relation between a source model and a target model is as follows:

```
relate MyRelation(s:S source, t:T target)
{
   equals(source.name, target.name),
   MyOtherRelation(source.value, target.value)
}
```

Listing 1 - 8 A model relation in MTF

The relation MyRelation has two arguments; a source model of type S and a target model of type T and has relation mappings defined in the body. The MTF does not specify which argument is input or output. If one is empty, the MTF will automatically fill out information so that expressions inside the relation are satisfied. For example, the equal expression states that the source name element is equal to the target name element. If the target element name is null, it will be filled with the source name. If it is not, the mapping is not satisfied and the discrepancy will be kept in a reconciliation log. The relation can also compose other relations so that relations and mappings can be decomposed and common relations can be reused. The MTF uses Eclipse Java Emitting Template (JET) to generate Java source code.

The IBM MTF is similar to our approach in supporting models based on various modeling languages. Input and output models must have types (s of type S and t of type T) while models in our approach must conform to modeling languages. However, they are dissimilar in that MTF uses a declarative language to transform one model to another and uses template for model to code while our approach uses Java for both transformations.

1.5.7 Parallax Framework

Another research work is based on aspect oriented approach. Silaghi's PhD thesis to be published in 2006 currently defines a model transformation based on concern weaving [Sil06]. His Parallax framework (Figure 1 - 11) uses aspects to weave concern oriented code that refines PIMs into PSMs. The transformation language is based on a declarative language called MTL [Fre05]. The framework defines four layers of concerns as follows:

- Middleware, e.g., distribution, concurrency, transaction, security etc.
- Technology, e.g., RMI, EJB/J2EE, CORBA, .NET, Web Services, Messaging etc.
- Platform products, e.g., WebSphere, WebLogic, JBoss, JOnAS, Axis, JMS, MQSeries, MSMQ, etc.
- Programming language, e.g., Java, C#, C++, C, Smalltalk, etc.

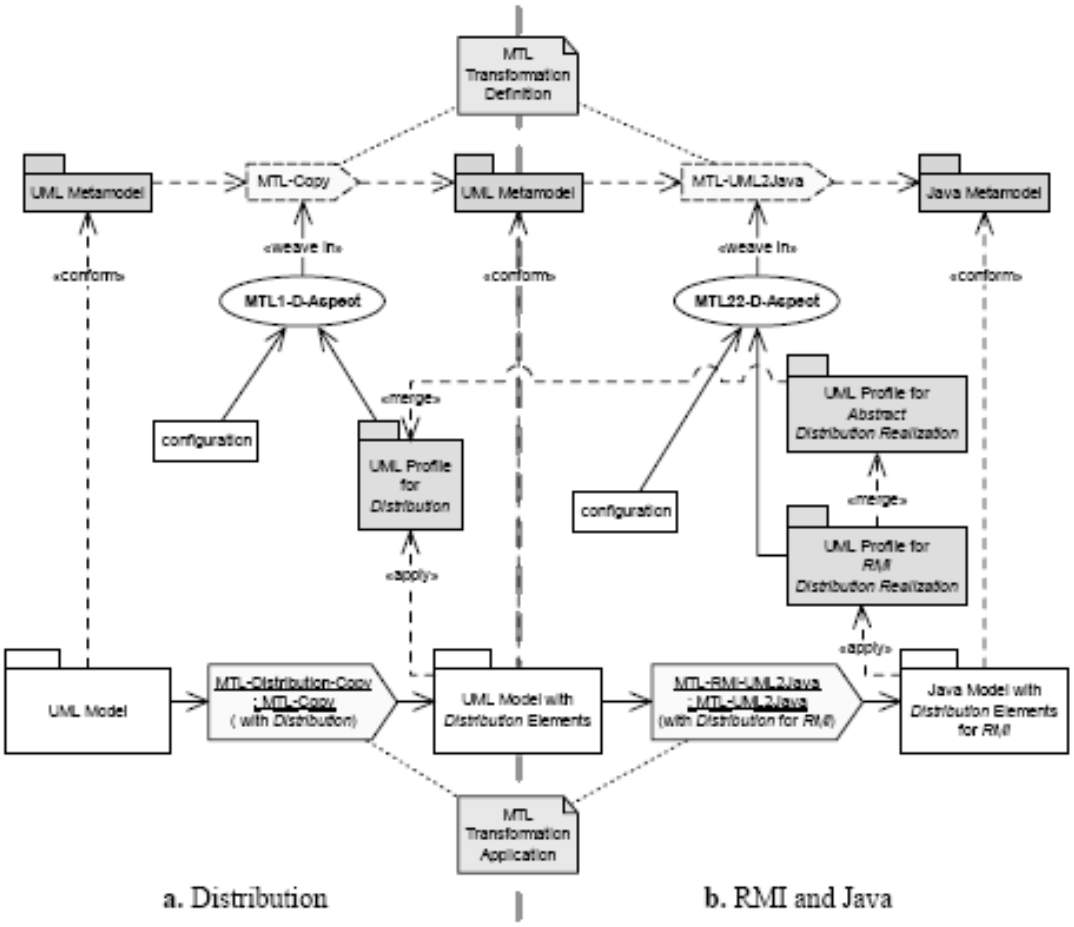


Figure 1 - 11 Distribution concern in Parallax Framework

A middleware concern may use different technology. For example, a distribution concern may use RMI, CORBA IDL or SOAP messages. Each technology may be implemented in different products. Developers can also use different programming languages. An aspect weaver must be defined for each middleware, e.g., a *DistributionAspect* for a distribution service and for each technology in the middleware, e.g., *DistributionSunJavaAspect* for a Sun Java RMI distribution. There is an aspect for each code generation, e.g., *JavaCodeGenerator*. The framework is similar to our approach in that it defines profiles for each middleware concern and uses Java to transform models. There are many similarities between our approach and Parallax. While our approach uses stereotypes to trigger a transformation, Parallax uses aspects to define pointcuts (stereotypes that apply to model elements) and advices (Java code that transforms the model). Our approach defines one profile for each middleware concern and store parameters specific to each technology in annotation files. Parallax defines one profile for each middleware concern and within a middleware concern. It defines a separate profile for each technology. Figure 1 - 12 below shows an example of a distribution profile [SFS04]. The profile has an internal structure defined in an abstract distribution realization profile. A concrete technology profile may extend ("merge" in UML terminology) from the abstract realization profile. The figure does not show a concrete platform product that implements the technology however.

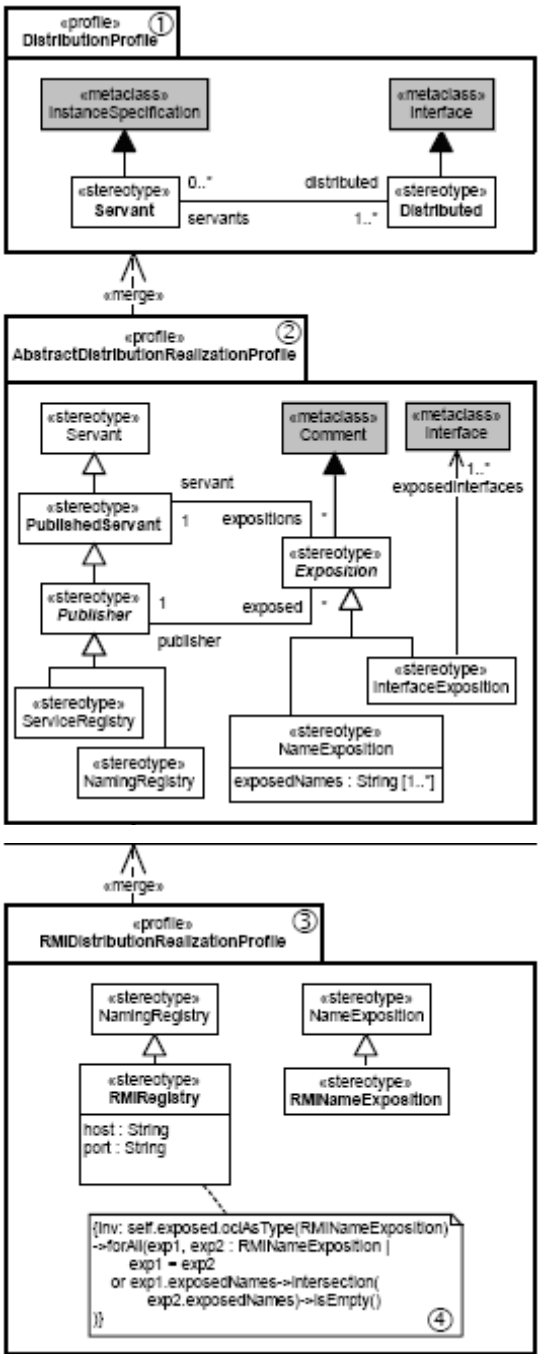


Figure 1 - 12 Distribution Profiles

Since the Parallax framework claims to support both middleware and language independent, it would be interesting to see how the framework takes care of technology that is based on a particular language such as J2EE/Java, .NET/C#. If a model is defined in one language and uses technology that is based on another language, how a

transformation deal with this situation. Another important question is how can the framework weave concerns that are not orthogonal, that is, they depend on each other. Does it mean that a weaver in one concern must be aware of another? If yes, how do they interact? For example, a transaction service usually concerns with persistent objects participating in a unit of work. While persistent objects use a service provided by a persistence profile, how a persistence aspect weaver obtains a transaction lock from the transaction aspect weaver? We would be interested to see these issues addressed in the author's dissertation. Our approach does not separate services as aspects because we believe that common services are not only used by application models but also by other middleware libraries. Even though we define services in separate profiles and store profile specific information in separate annotation files, a transformation code has a facility to query and update information inside a model as well as annotations from other profiles. It is therefore possible for a transaction transformer to modify an output model generated from a persistence transformer to return a transaction lock instead of a default persistence lock.

In addition to the generative approach, some research focuses on verification and executability of the model. Baar stressed the importance of formal semantical foundations of all languages to be used to express executable models and classify symbolic conformance tests for implementation models [Baa02]. Graw presented a method to verify xUML specifications in the context of MDA by using a variant from Lamport's Temporal Logic of Actions [GH02]. Riehle developed a UML virtual machine that allows models to be executed [RFBO01]. Gallardo provided a model checker to debug design from UML models [GMP02].

Our transformation framework, *Mercator* [WJ04] is based on object-oriented, extensible, stereotype-based model transformation that supports pluggable object service transformation. Mercator differs from the others in that it defines not only an object oriented transformation framework but also middleware independent APIs that modelers can use in their platform independent models. Mercator does not invent another transformation language. Instead, it uses XML to define profiles and Java to transform

models. By using well known languages like XML and Java, it may have fewer barriers to entry from IT shops that are already familiar with these languages. Mercator is also designed to allow model compiler developers to extend the framework to support new profiles and transformations.

This dissertation defines a standard way for model compiler developers to develop model transformation and refinement for PIM and platform independent service libraries that model designers can use. One company may reuse a domain specific PIM and generate customized implementations for their needs by developing their own library transformation code. Another company may buy new transformation code from vendors to migrate their system from one technology to another. We believe that a platform independent library specification and the model transformation framework will have an impact on the reusability and standardization in the model driven development.

1.6 Thesis Statement

One of the reasons enterprise software is difficult to change is because it relies on middleware services. Middleware makes the rest of the application simpler but then the application depends on the middleware. Migrating existing software to new middleware technology requires significant software redesign, rewriting and testing.

By specifying common services using UML profiles that do not depend on specific middleware technology and providing separate transformations for each service, it is possible to develop business software models at a middleware independent level and use transformers to map the models to middleware specific implementations.

This dissertation defines these platform independent libraries using the standard extension mechanism of UML profiles. These profiles consist of stereotypes, stereotype attributes and APIs. Models can import the profiles, mark model elements with stereotypes and annotate the stereotypes with attribute annotations. A model transformer framework must support model transformations from models that use these profiles for existing object services as well as a standard way to define new services.

The Mercator model transformation framework provides model compiler developers a systematic way to define middleware independent services as well as model manipulation APIs to refine model elements that use these services. Information specific to particular middleware technology is customizable and kept outside of platform independent models.

Our contributions in this research are:

- A systematic method to define middleware independent object services.
- Profiles for object persistence, naming, distribution, transaction and messaging services. These profiles contain stereotypes, APIs, annotations and model transformers.
- A novel, lightweight, stereotype-triggered model transformation framework that allows object service transformers to plug-in to manipulate model elements.
- A modeling language based on eMOF with support for UML profiles and action semantics.
- A Mercator tool that implements the framework.

We believe that this research is a step towards the realization of the MDA approach.

Chapter 2 Mercator Model Transformation Framework

2.1 Model Transformation

A model transformation is a function of an input model and a set of parameters and returns an output model. Input and output models must conform to their modeling languages.

$$M_o = T(M_i, A)$$

where

- T = Model transformation function
- M_i = Input model that conforms to a modeling language ML_i
- A = Annotation that contains a set of parameters used for T
- M_o = Output model that conforms to a modeling language ML_o

M_i and M_o do not have to belong to the same modeling language. However, modeling languages should conform to a standard framework so that models created from these modeling languages can be manipulated in a standard way. One such framework is the Meta Object Facility (MOF) standard from the OMG. The MOF is a general framework for describing object oriented languages. It is a centerpiece in the OMG four layer modeling stack [Mof03] where a model in one layer is an instance of another model in an upper layer. Bezivin used an analogy between the modeling stack and one from programming languages to explain model relationship between layers [Bez01]. In Table 2 - 1, at the lowest level, M0, objects are corresponding to an execution of a program in Java. At level M1, objects are instances from UML classes while the program execution is run by a Java program. At level M2, the UML classes conforms the UML notation similar to the Java program conforms to the Java grammar. At level M3, UML is one among modeling languages described by the MOF. It is similar to the Java grammar that is defined by an EBNF. However, MOF focuses mainly on defining object oriented

languages while EBNF is more general and can be used to define any programming language.

Level	*Modeling paradigm*	*Programming paradigm*
M3	MOF	EBNF
M2	UML	Java grammar
M1	UML model	Java program
M0	Objects	Execution of Java program

Table 2 - 1 OMG four layer modeling stack

Since there are two main levels of abstraction in the MDA; a platform independent level and a platform specific level, there must be two transformations; one for PIM to PSM and the other for PSM to code.

PIM-to-PSM: $M_{PSM} = T_{PIM\text{-}to\text{-}PSM}(M_{PIM}, A_{PIM\text{-}to\text{-}PSM})$

PSM-to-Code: $M_{Code} = T_{PSM\text{-}to\text{-}Code}(M_{PSM}, A_{PSM\text{-}to\text{-}Code})$

At each level, models use middleware services such as naming, persistence, transaction and messaging. These services at PIMs should not contain implementation details while ones at PSMs must have complete information so that a concrete implementation can be generated. While it is natural to include transformation rules for all services into the $T_{PIM\text{-}to\text{-}PSM}$ and the $T_{PSM\text{-}to\text{-}Code}$, the result does not scale well. It is difficult to add new middleware service to the transformations especially when each service is created by different companies. We believe a transformation based on individual service that is independent is a better approach. Each service should have its own transformation and the transformation should be as independent as possible. For example, a PIM that uses a persistence service and a messaging service is transformed by each service transformer and by the $T_{PIM\text{-}to\text{-}PSM}$

$$M_{PSM} = T_{PIM\text{-}to\text{-}PSM}(T_{nessaging}(T_{peristence}(M_{PIM}, A_{PIM\text{-}persistence}), A_{PIM\text{-}messaging}), A_{PIM\text{-}to\text{-}PSM})$$

The order of the transformations is important. A service that uses another service must be translated first. The relationship between services must be captured in a profile. The transformation function must be composable so that a transformation function can take an output model from another. In the end, the $T_{PIM\text{-}to\text{-}PSM}$ transforms the intermediate model that has been transformed by all services into a PSM.

However, since the transformations can be chained, they assume that input and output models must be at the same level of abstraction. On the other hand, the model still cannot contain any middleware specific information until the $T_{PIM\text{-}to\text{-}PSM}$. The annotation model such as $A_{PIM\text{-}persistence}$, $A_{PIM\text{-}messaging}$ only stores information about the PIM with regards to the persistence and messaging service respectively. Therefore, our framework adds an intermediate model, MPIM, to keep the result of the PIM transformation and be an input to the PSM transformations. The MPIM is a kind of PIM that contains partial information of services and can be composed by transformers. The idea of intermediate models is similar to the abstract platform proposed by Almeida et al [Alm05]. In addition to a PIM service transformation, each service also provides a set of PSM transformations. Developers can choose service transformations from middleware specific choices.

$$M_{MPIM} = T_{PIM\text{-}to\text{-}MPIM}(T_{PIM\text{-}messaging}(T_{PIM\text{-}peristence}(M_{PIM}, A_{PIM\text{-}persistence}), A_{PIM\text{-}messaging}), A_{PIM\text{-}to\text{-}PSM})$$

$$M_{PSM} = T_{MPIM\text{-}to\text{-}PSM}(T_{PSM\text{-}messaging}(T_{PSM\text{-}peristence}(M_{MPIM}, A_{PSM\text{-}persistence}), A_{PSM\text{-}messaging}), A_{MPIM\text{-}to\text{-}PSM})$$

Since the number of services is not fixed, the design of a model transformation framework should be flexible enough to support new service transformations as well as existing ones. The Mercator framework supports an arbitrary number of service transformations. Each service is described by a profile. A profile consists of stereotypes and APIs that extend the model elements to have service related capability. The Mercator framework associates each stereotype with a set of transformations that maps a

model element that is marked with this stereotype into an output model element. Modelers choose which transformer should be used for each model element. Otherwise, a default stereotype transformer will be used. Unless, the stereotyped model element contains error(s) or needed to be transformed with the same transformer again, the transformer must change or remove the marked stereotype so that the same transformer will not be triggered again.

At the top level, a root node of each model must be marked with «PIM», «MPIM», or «PSM» which are stereotypes defined in the Mercator profile. The framework includes transformers that map the model from PIM to MPIM, MPIM to PSM and PSM to code files respectively. There can be more than one transformer for each stereotype. For example, a testing transformer that generates additional instrumented model elements and a production-level transformer that generates optimized outputs.

It is important to note that each profile contains stereotypes related to the service provided by the profile. The profile itself does not specify how these stereotypes are mapped into implementation. Mercator is a framework that associates transformers to stereotypes and specifies transformations for model elements that are marked with these stereotypes. New transformers for each stereotype can be added without changes to the service profile. Since the profile definition and the transformation definition are separate, it is possible to use profiles with another transformation framework and still achieve the middleware independent modeling goal.

Before we go into the framework details, we need a standard way to represent models. The OMG defines two MOF specifications; an essential MOF (EMOF) and a complete MOF (CMOF). The former is a basic modeling language that can be used to describe object oriented languages while the latter adds UML specific elements and a reflection capability to describe the UML in particular. We found that the EMOF is generic and sufficient to describe modeling languages in Mercator. The EMOF model elements are described in Appendix A.

However, the MOF does not have a powerful extension concept like the UML profile. Each element can only have zero or more string comments and there is no semantics for these comments. The UML profile is a powerful concept that uses stereotypes to extend model elements with new structure and/or behaviors. Stereotypes are names enclosed in guillemots. For example, «persistence» is a stereotype, and in this dissertation is used to mark a Class model element whose instances are stored in persistent storage. A stereotype has constraints that must be satisfied. These constraints consist of a precondition to indicate which model elements can be marked by the stereotype; a postcondition that specifies which extra capability the model elements have after the stereotype is applied. A «persistence» class has persistence methods to load, store, lookup and delete objects. These extra methods are persistence APIs that application code can call to use the persistence service.

Stereotypes have no specific semantics and don't contain information about implementation. They do not tell how the system stores objects from the persistence class. This is fine when we deal with models at a platform independent level. However, to generate a concrete implementation, we need to specify how stereotyped model elements are implemented. There are many ways to achieve this. One way is to specify the implementation method within the stereotype as a stereotype attribute. For example, a «persistence» stereotype could have an attribute *method=XML*. A stereotype can contains many attributes, similar to a class can contains properties. We use «persistence { method=XML }» to indicate the stereotype and its attribute. If there are more than one attribute, we use commas to separate them.

Even though we could specify a persistence method as a stereotype attribute, it is not practical to do for each persistence class in a large model. Instead, we store common information in a separate location called an annotation. A persistence annotation contains default attributes that apply to all persistence classes. Unless the model developer overrides the attribute, the transformation tool will use the default value from the annotation. For example, we can indicate a system wide persistence method to relational tables while some objects from specific persistence classes will be stored as XML files.

One of the biggest advantages of having separate annotations and not adding information directly to the model is that it is easier to change from one middleware package to another. The annotation contains all the information that will change, so we have separated changeable from nonchanceable information.

2.2 Extending EMOF

We extend the EMOF with the UML profile extension so that modelers can apply UML stereotypes into EMOF element. We come up with a simple abstract syntax structure that is used to construct models and elements within a model.

A model abstract syntax tree is a data structure that can be used to describe a modeling language (M2). The tree consists of an abstract syntax node with the following properties:

- Composite. A node is either a terminal node or one that contains children nodes.
- Node name. Each node has a name. Its fully qualified name is derived from a concatenation of all its parents' names and its name.
- Attribute pairs. A node contains a list of named-value pairs.
- Stereotypes. A node contains a list of valid stereotypes. Valid stereotypes are from profiles imported in the model.
- Factory: A factory object that creates this node.
- Viewpoints. A node contains a named list of viewpoints. Modelers can choose to view model elements in a specific set of viewpoints and hide ones that are irrelevant to current task at hand.
- Exceptions. A node may be invalid during editing and transformation. A model editor or transformers add exceptions to the node when the node violates constraints imposed by stereotypes.

There are 10 different basic model elements subclassing from the abstract syntax node; model, package, class, property, operation, library import, profile import, parameter,

association and association end. Other elements are constructed from these elements. For example, an interface element is a class element with an *isAbstract* property.

The factory object is a key element. It is responsible for constructing, storing and loading model elements for a given language. Each modeling language has one factory. For example, a Java factory object is responsible for models written in Java while a UML factory object is for models in UML. This way, the Mercator tool uses the factory objects to load and store models from arbitrary language. Model transformers call the factory object to create new model elements of that language. A model transformer can be developed to map an input model in one language to an output model in another. The framework is not limited to specific languages. It supports new languages by adding a new factory object and developing transformers for those languages.

A model transformation is therefore a function that maps an input model tree of a factory object into an output model tree of another factory object. Extra information used during the transformation is stored into separate annotation files. Each file contains annotations for each middleware service. An annotation is a hierarchical tree in which each node has a name and a value (n, v). A non-leaf node contains specialized information specific to the node. For example, a PersistenceMethod node has a default transformer to EJB that applies to all persistence classes. However, modelers can override some persistence classes to use an XML transformer to persist objects for specific classes or classes in specific packages.

2.3 Model Representation

A model represents something. In software, a model is a view of a certain aspect of a software system. It can be described textually as a programming language code or visually as a graphical notation. Either way, it must conform to a language. A programming language code conforms to its language specification while a graphical notation conforms to its modeling language. Internally, both can be represented as an abstract syntax tree (AST). Intentional programming [Sim96] uses this approach and represents the tree in various forms that programmers or end users can edit.

The OMG proposes the UML as a universal modeling language [UML05]. It current version of the specification is 2.0. The UML is a large specification because it tries to serve many different purposes to many different groups of modelers. The current UML 2.0 specification contains 13 diagram notations in 3 groups; structure, behavior and interaction. It also has a declarative object constraint language (OCL) to specify invariants and it has action semantics that is as expressive as programming language statements. One particular UML Infrastructure model element has 18 ancestors and they can not be easily shown together in one diagram. To understand the semantics of this element; we have to look at them from many diagrams. The large and complex dependency is one of the factors that make UML imprecise and difficult to understand. Reggio and Wieringa documented the inconsistencies of previous UML specifications [RW99] and the Eclipse UML2 implementation also discovered many inconsistency issues from the current UML 2.0 specification [EUML2]. Kobryn documented several areas that UML2.0 still falls short [Kob04] and emphasized the lack of UML reference implementation and test suites made vendor tool compliance nonexistence. It seems that the consistency goal of the whole specification may not be realized in a near future.

Since UML uses XMI to store models, one could use XMI as an AST. However, XMI is not only too verbose but its current library implementation [EUML2] does not allow model elements to be partially consistent which is required for in-place model editing. Mercator defines its own format that is compact and close to Java, its target generated language.

The UML uses the XML Metadata Interchange (XMI) to store visual models into textual files. Even though the XMI name suggests interoperability for exchanging UML models, it's hardly the case in practice. Because there are many versions of the XMI (there are 6 revisions from 1.0 to 2.1) and each version can be used to store different versions of UML (there are 7 revisions from 1.0 to 2.0), it is non-trivial to import particular UML models produced from one tool vendor to another.

We use Java programs as models. Java is popular and has a well defined meaning. Java programs can also be visualized as UML diagrams. Throughout the dissertation, we will show models as Java source code or UML diagrams depending on which viewpoint is more compact and meaningful to represent topics we discuss. For example, class diagram is better at expressing different kinds of associations and cardinality. To support the UML extension mechanism in Java, we extend the Java language notation to include the UML stereotype specification. Internally, we store models in an XML representation that can be exported into the latest XMI 2.1.

To illustrate the model representation of a class, suppose a Player class is defined in a class diagram as follows:

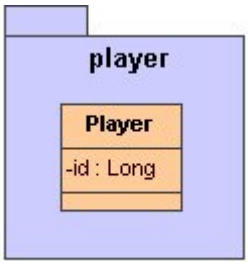


Figure 2 - 1 A Player class diagram

The above class diagram is represented in XMI as:

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:uml="http://www.eclipse.org/uml2/1.0.0/UML"
    xmi:id="_CBsuwNYFEdmtPNElXw9sgw" name="player">
  <packageImport xmi:id="_CBsuwdYFEdmtPNElXw9sgw">
    <importedPackage xmi:type="uml:Model"
        href="UML2.library.uml2#_kO9GNSRkEdmW25Ue05Bnnw"/>
  </packageImport>
  <ownedMember xmi:type="uml:Package" xmi:id="_CBsuwtYFEdmtPNElXw9sgw"
   name="player">
    <ownedMember xmi:type="uml:Class" xmi:id="_CBsuw9YFEdmtPNElXw9sgw"
     name="Player">
      <ownedAttribute xmi:id="_CBsuxNYFEdmtPNElXw9sgw" name="id">
        <type xmi:type="uml:PrimitiveType"
            href="UML2.library.uml2#_kO9GPyRkEdmW25Ue05Bnnw"/>
```

```
        </ownedAttribute>
    </ownedMember>
</uml:Model>
```

Listing 2 - 1 A model representation of the Player diagram

Mercator simply stores the same model as:

```
<pim:model xmlns:pim="http://mercator.org/pim" file="player.xml">
<import file="uml.xml" />
<package name="player">
 <class name="Player">
  <attribute name="id" type="UML::Long" />
</class>
</package>
</pim:model>
```

Listing 2 - 2 Mercator representation of the Player diagram

Another example shows an application of stereotypes and their attributes in Java code.

```
«consumer» class Supplier {
    «subscribe {channel=”supplierForThisMaterial”}»
    public ResultCode processRequest(«message»String text) {
            …
            ResultCode code = …;
            return code;
    }
}
```

Listing 2 - 3 A Java source code with stereotype annotation

Which is equivalent to UML class diagram notation in Figure 2 - 2 below.

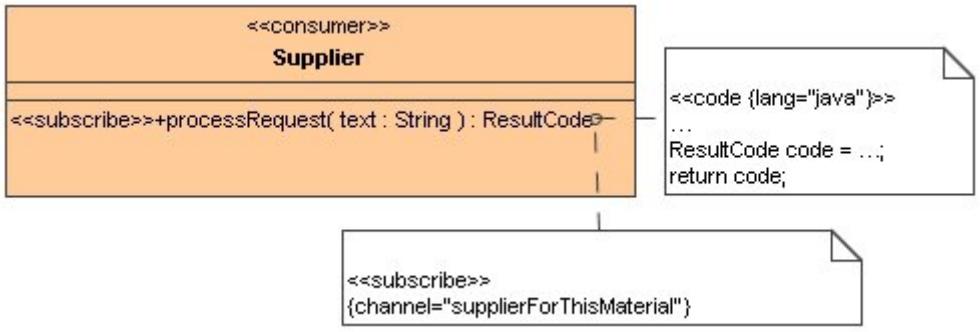


Figure 2 - 2 UML Class diagram of the Java source code.

Keep in mind that Mercator does not manipulate diagrams or Java code but actually an XML representation. This notation is similar to Java 5.0 annotation feature except that

the Java annotation applies only to packages, types, methods and fields while the UML stereotype is more fine grained and applies to any modeling element. The above example shows that «message» applies to a method parameter. Some stereotypes apply down into statements in method body. Therefore, if we were to use Java annotation, the annotation application must be relaxed so that we can annotate Java code down into Java method body, method parameters and statements.

2.4 A Stereotype-triggered Model Transformation Framework

2.4.1 Introduction

Mercator uses stereotypes to control the transformation. Each stereotype has one or more transformers that are stored in a transformation registry. A stereotype may have more than one transformer if there is more than one way to transform the stereotype.

There are two kinds of transformers: a node transformer and a tree transformer. A node transformer takes an input model element, validates the input and produces an output model element. Most transformers are node transformers. A tree transformer takes an input model tree, iterates over the nodes in the input model tree in preorder, obtains stereotypes for each node and looks up node transformers from the registry and invokes these node transformers' *transform()* methods with the current visiting node as the parameter. If the node element does not contain a registered stereotype, it will be copied unchanged to the output model.

A node element may contain more than one stereotype, for example, a class that is persistable and distributable. Mercator puts the corresponding transformers into a transformation queue. The order of the transformation is important because a transformer from one service may check properties generated by another transformer. For example, the distribution service uses the persistence service to serialize objects into streams. If the distributed objects are not persistable or the persistent method produces objects that are not streamable, they can not be distributed.

Mercator usually iterates several times over a model. It will end when the model no longer contains transformable stereotypes. After all nodes are visited and no transformation error occurs, the output model tree will be given another stereotype so that the transformation algorithm will be applied again with the next tree transformer of the new stereotype. For example, the PIM-to-intermediate-PIM tree transformer takes a «PIM» input model, creates an output model and assigns the «MPIM» to the output so that Mercator will look up the intermediate-PIM-to-PSM transformer matching the «MPIM» from the registry and invoke it during the next transformation iteration. The transformation process will stop when the output model tree is no longer stereotyped which is usually a case after the PSM transformer generates source code from the PSM.

A node transformer validates the input node element and may look into its subtree to makes sure that the input node and its children meet a set of constraints, for example, a persisted class (class that is stereotyped with «Persistence::persistence») must contain an attribute that is stereotyped with «Persistence::id». Mercator provides basic validators such as type validator, owned element's stereotype checking, naming rules and allows model compiler developers to define new ones.

The transformer may store or consult additional information from an annotation data structure. This data structure stores transformation related parameters and lives over the entire transformation process. At first, the annotation file for each profile is created from a profile annotation template. This annotation file can also be stored and used in subsequent transformation. If the required information does not exist in the annotation, the node transformer will generate an error, thus preventing the tree transformer from proceeding. The model designer has to check the error, specify the required information and rerun the transformation tool again.

Mercator performs model transformation in 3 passes using different tree transformers: 1) PIMTransformer traverses a PIM, invokes platform independent stereotyped node transformers and generates an intermediate PIM, 2) MPIMTransformer transforms UML model elements in the intermediate PIM into Java elements in a PSM, maps UML

primitive types into Java types and invokes platform specific stereotyped node transformers to generate the output PSM. This output is a horizontal refinement of the input model with the expanded semantics of the applied stereotypes. 3) PSMTransformer generates Java source code.

Model compiler developers extend Mercator by creating node transformers for middleware independent libraries and register them with the transformation registry. Each library stereotype contains at least 2 transformers; one for PIM-to-MPIM and the other for MPIM-to-PSM. These transformers will be invoked when the tree transformer visits the model elements that use the library stereotypes.

During the model transformation, model designers may have to specify extra information in the annotation file. For example, there are many supported object persistence methods. The model designers have to choose a persistence method, data source type and an object identity type. Choices in persistence method may be among EJB, Hibernate, JDO, and XML etc. Data source type may be database, network streams or files.

2.4.2 Transformer Queue

A model usually contains many stereotyped nodes. Each node may have more than one stereotype and each stereotype has an associate transformer that accesses the node data structure. Some transformers need to access properties produced by others. Therefore, when a tree transformer visits tree nodes and finds stereotype transformers, it will not invoke them immediately. Instead, each transformer will be put into a transformer queue and the order of the transformer depends on the dependency definition of the profiles. Transformers that depend on a result of another will be put after that transformer. The transformer queue element contains a triple of a transformer instance, a stereotyped node and the annotation tree. After the tree transformer visits all nodes and collects all transformers, it will iterate the queue, obtain the stereotyped nodes and the annotation trees and invoke the transformers. If there are transformation exceptions during a transformer's execution, it will collect the exception that consists of the error node and error description into an exception queue. If there is no exception in the exception queue,

the tree transformation will change the model type from «PIM» to «MPIM» or from «MPIM» to «PSM» or from «PSM» to null. If the model type is not null, the framework will find the tree transformer of that type and invoke it. The transformation process ends with the model is not stereotyped.

2.5 Transformer Development Use Cases

There are two primary use cases for model compiler developers. The first is to develop a new library profile. The goal of Mercator is to define commonly used platform independent object libraries. However, there may be a need for new libraries for use in specialized problem domains. For example, a B2B parts order profile for automobile supply chain system may require a specialized specification of parts and inventory. Model compiler developers define a set of related stereotypes in a library profile. Each stereotype contains properties and a specification of platform independent APIs. These APIs are contracts that can be used in a PIM and that each transformer must implement. Each stereotype also contains at least one transformer each for a PIM-to-MPIM and MPIM-to-PSM transformation and provides a way to add new transformers to generate different implementations. A library profile should contain test suites that can be tested during new mapping development.

The second and probably more common use case is to implement mappings of an existing platform independent library profile into a new implementation. For instance, model compiler developers may want to map object distribution using specialized object transport method instead of the provided Java-RMI or SOAP/XML. They will reuse the Object Distribution profile and create new mappings that generate different implementation from the defined platform independent APIs. For each defined stereotype, developers implement the required validator and transformer's methods. Most of the time, they would extend from AbstractTransformer for transformation that occurs only at the stereotyped model element. For validation, developers can choose among available validators or develop custom validators by implementing from the IValidator interface. The other kind of transformation is a tree transformer. This kind of transformer extends from AbstractModelTransformer and is used for transformations that

require an entire model tree traversal. For example, a PIM-to-MPIM, a MPIM-to-PSM and a PSM-to-Code iterate over the entire model tree, look for stereotypes in each model elements and put stereotype transformers into a transformer queue. The queue is ordered by the dependency of each stereotype dependency, for example, a message stereotype uses persistence service to stream objects over a network, the persistence message object uses naming service to assign a name to the message so that both sender and receive ca use the name to publish and subscribe the message. The transformers for each service will be put in a dependent first order. In this case, the naming transformer is put first, followed by the persistence transformer and then the messaging transformer. At the end, the tree transformer iterates the transformer queue and executes the transformers in the queue one by one starting with the input model. Each transformer returns an output model that will be used as an input to the next transformer.

Developer registers the transformers into a pool of available transformers in the Mercator transformer registry. Each entry contains the stereotype name, the model it is applicable to (PIM, MPIM or PSM), a set of transformer class names and a default transformer. If the modeler does not specify a transformer for each service, a default transformer of that service will be used. Finally, developer runs the test suites defined in the library profile against new mappings to ensure that the result of the transformation works as expected.

2.6 Summary

This chapter provides a theoretical foundation for composable model transformation in the presence of annotation parameters. It describes an extension to the OMG's standard modeling language eMOF to support stereotyping and model factories among others. Next it shows an object oriented model transformation based on stereotype triggers. The framework defines a standard way to create transformable profiles and a tool that allows model designers to create models, import and use library profiles [WJ04]. A profile consists of a set of related stereotypes and a set of middleware independent APIs. Modelers import profiles and use the stereotypes defined in these profiles in the PIMs. PIMs use the middleware independent APIs and do not have to concern how the APIs are implemented. Model compiler developers are responsible for creating transformers for

each middleware product. Model transformation usually requires extra information specific to a particular middleware. We store this information into separate annotation files that can be customized and reused. The Mercator tool provides user interfaces, basic model element validation, traversal, UML-to-Java mappings and supports pluggable transformers. Transformers can be added to generate more efficient code and/or to support new target middleware. Thus, it becomes possible to use the framework to generate an implementation based on one middleware to another by choosing a different set of transformers to apply to the same PIM. An implementation of this framework is described in chapter 8.

Middleware independent service support in Mercator provides a higher level of abstraction that decouples business models from middleware specific information and allows application modelers to model systems with high level enterprise patterns such as Patterns of Enterprise Application Architecture [Fow03], Domain Driven Design [Eva03], business patterns and objects [EP99].

Chapter 3 Persistence Service

3.1 Introduction

Data persistence is an important part of businesses; online stores keep customer records, banks maintain account transaction history, marketing departments track customer relationship information. These data must be kept permanently so that they can outlive the process that created them and can be accessed later. In object oriented systems, data are stored inside objects and we call a mechanism of storing and retrieving in-memory objects to and from secondary storage *object persistence*. Persistence can be complicated since objects often have relationships with one another and storing them becomes storing their object graphs with the objects at the roots and nodes are other objects reachable from the parent nodes. There are also many ways to represent objects in secondary storage; they can be flat files, hierarchical data structure or relational data. Data integration between systems that use different representation of the same data becomes non-trivial and error prone. Companies that have invested a huge amount of money in one persistence technology often resist to change to another because the persistence layer is tightly coupled with their domain models.

There are many persistent middleware package [JDO04] [Hib06] [Top06]. Each middleware has its own strengths and weaknesses and there will probably never be one that suits every need. This makes it hard to design systems that are independent of persistence middleware because each middleware introduces complexity from its APIs. Once a domain model is designed specific to APIs from one middleware, it would take a big effort to refactor the model to support other persistence libraries.

This chapter introduces a middleware independent persistence profile that allows for modeling object persistence in PIMs. The profile consists of a set of persistence stereotypes, middleware independent persistence APIs and a set of transformers that

generate implementations based on persistence choices and annotation parameters supplied by modelers.

3.2 A Motivating Example

Suppose a car company builds a vehicle storage object model. A Vehicle class is a base class that contains basic information about the vehicle. There are many subclasses of Vehicle; namely, Automobile, Truck, SUV, Bus, RaceCar etc. Each vehicle may be kept in a primary garage. Each garage has a certain capacity and cannot keep more vehicles than its capacity. Figure 3 - 1 shows a class diagram.

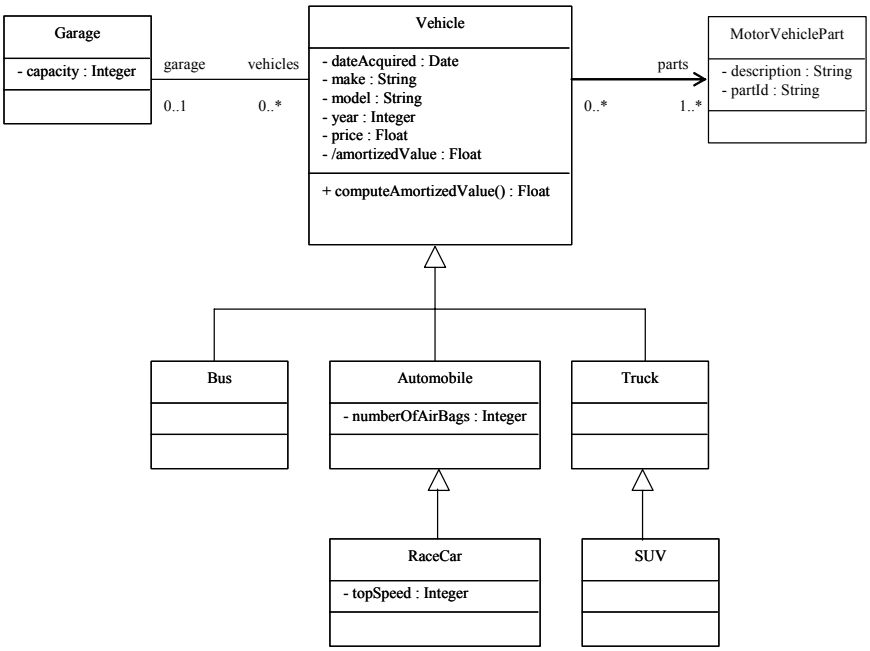


Figure 3 - 1 Vehicle class diagram

To create an instance of a vehicle and store it, we would like to do something like this:

```
Vehicle car = new Automobile();
car.setNumberOfAirBags(2);
…
car._store();
```

However, if we choose a Hibernate persistence library, an implementation would look like this:

```
try {
   Session session = HibernateUtil.currentSession();
   Transaction tx = session.beginTransaction();

   Vehicle car = new Automobile();
   car.setNumberOfAirBags(2);

   session.save(car);

   tx.commit();
   HibernateUtil.closeSession();
} catch (HibernateException e) {
   e.printStackTrace();
}
```

The *HibernateUtil* is a helper class that creates, open and close sessions from a configuration file that describes class relationship and its mapping to relational tables.

To find all instances of *Vehicle* class hierarchy, we would like to call a class method *findAll(true)* where the true parameter indicates all instances of the vehicle class including instances from its subclasses.

```
List cars = Vehicle._findAll(true);
```

The Hibernate implementation becomes:

```
try {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        List cars = session.createQuery("from Vehicle").list();

        tx.commit();
        HibernateUtil.closeSession();
} catch (HibernateException e) {
        throw new RuntimeException(e.getMessage());
}
```

Once a persistence object is created, a persistence service gives it a unique identifier so that a program can use that identifier to retrieve the object from the persistence store. Persistence objects contain an additional method *_getId() : String* that must return a unique identifier string. This identifier can be a natural key or a surrogate key assigned by the persistence service. The developer uses this identifier to find a specific object.

```
car._store();
String id = car._getId();
…
Vehicle car = Automobile._find(id);
```

To destroy the object and remove it from the persistence storage, we should simply call:

```
car._remove();
```

Once the object is removed from the persistence store, the object will live only in the main memory.

Persistence middleware such as Hibernate requires configuration files that describe class structure and relationship as well as how each class maps into files or database tables. Since the class diagram in Figure 3 - 1 contains class relationships, it is relatively simple to automatically generate these configuration files.

However, if the developer changes the persistence library to another such as EJB or XML, he has to change the structure in the code. Persistence classes inherit from different superclasses. Session objects are different. So are persistence-related methods. This becomes tedious and error prone. Instead, if the developer defines class relationship in the class diagram and writes code according to the middleware independent APIs, Mercator will transform the model in the class diagram into different implementations. Parameters specific in a persistence middleware will be kept separately in each annotation file.

3.3 Object Persistence Characteristics

Persistence has five primary characteristics that a profile must address.

3.3.1 Object model

A class diagram is a common way to define object structure and relationships with one another. It contains information about object graph reachable from the parent object (persistence by reachability). Object oriented programming languages use classes to describe objects. However, untyped collection fields make it difficult to infer the types. Prior to Java 1.5 generics, Java developers could not specify types of collection fields. The specification of object structure and relationship are often described using specific languages. For example, the ODMG uses the Object Definition Language (ODL) while an RDBMS uses the Data Definition Language (DDL). Since the RDBMS does not store objects into relational tables directly, there must be mapping files to facilitate object-to-relational data translation. Some persistence libraries use mapping files to automatically generate DDL files. However, the object-to-relational impedance mismatch problem is a hard problem [SZ87] [MBB06] [BGD97]. Developers must be careful especially when mapping class inheritance and polymorphism. Platform independent persistence service uses information from class models to describe object structure and its relationship. Model transformers use this information to generate configuration files and use middleware specific parameters to customize the configuration.

3.3.2 Object store representation

Different persistent middleware stores in-memory objects into persistence storage differently. Java object serialization mechanism uses proprietary binary flat-file format. Java 1.4 and above provides XMLEncoder and XMLDecoder classes to write and read objects into an interchangeable XML format. RDBMS uses table tuples to store objects. Internal table file formats proprietary to each RDBMS vendor. Some RDBMS vendors provide built-in support for objects. A platform independent persistence service should not require one specific representation. Developers should be able to use platform independent persistence APIs and not have to worry how objects are stored.

3.3.3 Object life cycle management

A persistent system must keep tracks of persistent objects. Oftentimes the system needs to free up memory by collecting objects that have been stored in a persistent storage. If those objects are needed again, they will be restored back to main memory. Persistent objects belong to two states; *passivated* when the state of the objects are stored in a persistent storage and the objects are safe to get collected and *activated* when the state of the objects are restored into memory and ready to use. In practice, these two states can be managed by two different strategies. The first strategy delegates the object life cycle management to a container. This strategy has an advantage that a container keeps track of persistent objects and can passivate and reactivate them as needed. This strategy is used by EJB 2.1. The other strategy put the life cycle management responsibility directly to objects. This strategy is used by lightweight frameworks such as Hibernate, JDO and Spring persistence. Objects contain persistent APIs to explicitly passivate and reactivate themselves. It is the responsibility of developers to write code that use these APIs to make sure that object states are saved in a persistent storage. Platform independent persistence service must provides APIs that can be transformed into either strategy.

3.3.4 Query language

Object querying is a powerful concept to find objects that meet required conditions. Queries are often used to look up objects from a persistent storage. Different middleware packages invented query languages. A direct JDBC implementation and Bean-managed persistence uses JDBC code to look up and convert data between objects and relational data while some frameworks such as iBatis and Spring provide lightweight SQL wrappers. Some frameworks introduce their own query languages [JDO04] [Hib06] [Sun06] [Cat97]. These languages are string based and usually interpreted and executed at run time. However, using string based query languages may not be effective because the query string may be constructed at run time similar to SQL. This makes it impossible to check for syntax or types at compile time. Most current refactoring tools do not support refactoring inside query strings. Cook and Rosenberger [CR05] proposed native queries as an alternative approach. Native queries use implementing language as the

query language. Listing 3-1 below shows an example of two query languages based on OQL and JDOQL and what native query in Java looks like.

```
OQL:
String oql =
"select * from student in AllStudents where student.age < 20";
OQLQuery query = new OQLQuery(oql);
Object students = query.execute();

JDOQL:
Query query =
persistenceManager.newQuery(Student.class, "age < 20");
Collection students = (Collection)query.execute();

Java Native Query:
List <Student> students = database.query <Student> (
   new Predicate <Student> () {
          public boolean match(Student student){
                 return student.getAge() < 20;
   }
});

C# Language Integrated Native Query (LINQ):
IEnumerable<Student> students = from s in students
where s.Age < 20;
```

Listing 3 - 1 Queries from different domain specific languages

Native queries are under active research. Microsoft introduced Language Independent Native Query (LINQ) in its .NET framework [Linq06] while Java 5.0 does not have this feature yet. One major criticism is a potential performance impact because the predicate evaluation retrieves all objects from a database to be evaluated at a caller side which may be expensive. Even though performance is not the focus of our research, we believe that one possible way to improve the performance when using native queries is to distribute the predicate code block and execute at the server.

Platform independent persistence service uses native queries as a standard way to retrieve objects.

3.3.5 Middleware specific APIs

Even though the persistence concept is similar, each middleware uses its own helper classes and persistence methods. Persistence managers have different names (SessionFactory in Hibernate, PersistenceManager in JDO, DriverManager in JDBC, EJBHome in EJB) while other frameworks do not have dedicated persistence manager (XMLEncoder or Java Serialization). Persistence methods are different (saveOrUpdate() in Hibernate, makePersistence() in JDO, executeUpdate() in JDBC, ejbStore() in EJB). Some have their own transaction support while others rely on global transaction manager or do not support transactions at all. Platform independent persistence service must provide unified APIs and a facility to check for transaction support. If a model requires transaction support but the selected implementation does not support transactions, the model transformation must fail and throw an error.

3.4 Middleware Independent Persistence Profile

The middleware independent persistence profile consists of a set of persistent-related stereotypes and stereotype attributes. Each stereotype contains a set of transformers; one for a persistence implementation. Each transformer has its own annotation file that stores information specific to this implementation. Domain models import the persistence profile by adding «import {profile=”persistence”}» to the model and use stereotypes to indicate class elements whose instances need to be persisted. By default, all class attributes will be stored unless they are marked by a «transient» stereotype. Listing 3 - 2 below shows a vehicle class with stereotype markings. Date and String primitive types contain objects whose values are stored directly. The *motorVehicleParts* is an ordered collection of the *MotorVehiclePart* type. How this collection is stored depends on whether the type is declared with «persistence» and will be explained next. The *amortizedValue* is a derived value and will not be stored.

```
«persistence» public class Vehicle {
   private Date dateAcquired;
   private String make;
   private List<MotorVehiclePart> motorVehicleParts;
   «transient» private Float amortizedValue;
}
```

Listing 3 - 2 Vehicle class definition

The persistence service identifies persistence objects by id. This identifier can be a natural id from the domain object or a system-assigned surrogate id. If it is the former, the developer marks the id field with «id». If it is the latter, the developer leaves out the «id» field and the persistence transformer will automatically assign one. There are several algorithms to assign unique identifiers. The choices of the algorithms are listed in the persistence annotation parameter. By default, the transformer uses a simple counter but the developer can override the algorithm globally or only for particular persistence classes.

When a class is marked with «persistence», the stereotype introduces 3 static methods; *_create(initialId : String) : Object*, *_load(id : String) : Object* and *_findAll(includeSubTypes : boolean) : Collection* and 3 methods; *_getId() : String*, *_store() : void*, *_remove() : void*. Developers may use different names for these methods. But if they do so, they have to mark them «create», «load», «findAll», «getId», «store», «remove» with respectively. The _create method instantiates an object. If the *initialId* is not null, the value is used for the object id. If it is null, the persistence service will assign one. The *_load* will retrieve the object from the persistence store. The *_findAll* will return an object collection of this type. If the *includeSubTypes* is true, it will also include all instances of its subtypes. The *_getId* returns a unique identifier string of the object. If the id field is not of type String, the transformer will convert it into a string. The *_store* and *_remove* will save the object into and delete it from the persistence store. Developers use these APIs to load, find and store objects without concerning how they are managed by a persistence store.

A PIM persistence transformer converts the code in Listing 3 - 2 into Listing 3 - 3. The expanded definition is shown in bold.

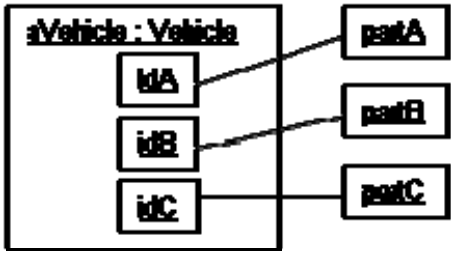
```
«persistence» public class Vehicle {
   private Date dateAcquired;
   private String make;
   private List<MotorVehiclePart> motorVehicleParts;
   «transient» private Float amortizedValue;

   «id» private String id;
   public static Object _load(String id) { … }
   public static Collection _findAll() { … }
   public String _getId();
   public void _store();
   public void _remove();
}
```

Listing 3 - 3 Expanded Vehicle class

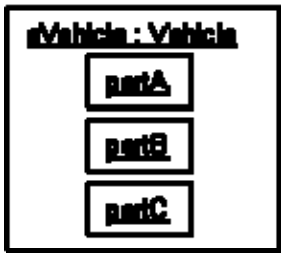
A vehicle object contains motor vehicle parts. If the vehicle is stored, how all parts are stored depends on whether the *MotorVehiclePart* is marked with «persistence». If it is, then they are each stored separately and only their identifiers are stored with the vehicle. Otherwise, their object graphs are stored along with the vehicle.

```
«persistence» public class MotorVehiclePart {
   «id» private String id;
   private String description;
   private String partId;
}
```



Listing 3 - 4 MotorVehiclePart is a persistence class

```
public class MotorVehiclePart {
   private String description;
   private String partId;
}
```



Listing 3 - 5 MotorVehiclePart is a value-object class

Persistence objects execute *_store()* synchronously. However, some persistence objects participate in a unit of work and their persistent states are stored only when the unit of work commits. Chapter 6 defines a specialized stereotype for persistent objects that participate in transactions. Table 3 - 1 below summarizes persistence stereotypes.

Stereotype	*Applies To*	*Description*
«import {profile=”persistence”}»	Model	Allows models to use the persistent profile.
«persistence»	Class	A class whose instances need to be persisted. If a class does not have the «persistence» stereotype, it is a transient object.
«id»	Attribute	A unique attribute that identifies each object. If a class contains more than one id attribute, they together constitute a unique id.
«transient»	Attribute, Class	A default property indicating that an attribute or a class needs no persistence.
«create»	Operation	A static operation that creates a new persistence object. If the operation name is _create(), this stereotype is optional.

«load»	Operation	A static operation that loads a persistence object from a data store. If the operation name is _load(), this stereotype is optional.
«findAll»	Operation	A static operation that returns all persistence objects from the class. If the includeSubTypes is true, it will return objects from subclasses as well. If the operation name is _findAll(includeSubTypes : boolean), this stereotype is optional.
«getId»	Operation	An operation that returns a string identifier. If the operation name is _getId(), this stereotype is optional.
«store»	Operation	An operation that stores the object into a permanent storage. If the operation name is _store(), this stereotype is optional.
«remove»	Operation	An operation that removes the object from a permanent storage. If the operation name is _remove(), this stereotype is optional.

Table 3 - 1 Persistence service stereotypes

3.5 Persistence Transformation

The Mercator persistence service transforms an input PIM in three steps. The first step expands classes in the input PIM with the persistence APIs. The second step takes the expanded PIM with a persistence choice and generates a PSM. The last step translates the PSM into source code. Each step, Mercator iterates over the model tree and activates a stereotype transformer associated with the stereotype. Some stereotypes contain more than one transformer. For example, an «id» stereotype contains simple counter, uuid, hilo transformers. The simple counter is the default but developers can override the transformer's choice.

Each transformer validates the model elements from a set of transformation rules. For example, if a class with «persistence» does not have an «id» attribute and the annotation property allows auto-generation, it will insert a default key attribute *id : String* into the expanded PIM. If the class already has an «id» attribute, the transformer uses it to return a unique id in the *_getId()*.

We will demonstrate two Mercator persistence transformers, JavaXMLTransformer and CMPEJBTransformer and how they are used to transform an object model into an executable model. Other possible implementation platforms for persistence include JDO, Hibernate, CORBA [Cor04] and JDBC [GJSB05].

The JavaXMLTransformer is the Mercator's Java-object-to-XML transformer based on JavaBeans' XMLEncoder introduced in Java 1.4. The XMLEncoder API uses JavaBeans' specification to introspect and build the object graph into an XML output while the XMLDecoder restores them back into objects. The JavaXMLTransformer transforms persistence classes into JavaBean classes by implementing the classes with the Serializable interface, creating a no-parameter constructor for each class and generating accessor methods for each persistent field.

The CMPEJBTransformer transforms a persistence object model into one using the Container Managed Persistence (CMP) 2.0 specification from the Enterprise JavaBeans (EJB) [Sun06]. We use three main features in CMP 2.0, the Container Managed Relationship (CMR) that specifies entity beans and their relationships, the EJBQL that provides a standard query language to obtain EJBs from a data store, and the local interfaces for objects that reside in the same deployed node.

In the next section, we will show that we can transform the same persistence-marked PIM into different platform specific models by means of applying mapping rules and annotating the model. The followings are steps in the persistence modeling and transformation.

1. Import the persistence profile into the object model.
2. Mark persistence classes with «persistence» and specify identifier(s).
3. Specify the persistence choice.
4. Customize the PSM with annotation parameters.
5. Generate an executable code.

3.7 Case Study

We will use an example from the motivating example section. We use Java and EJB metamodels as defined in the OMG's Metamodel and UML Profile for Java and EJB, v1.0 [Edoc04]. We assume that the system is deployed in a single address space. Otherwise, if objects reside in different address spaces, we need to use object distribution service to define object boundary. This use case will be described in chapter 5.

3.7.1 A JavaXMLTransformer

Step 1 – Import the persistence profile into the object model.

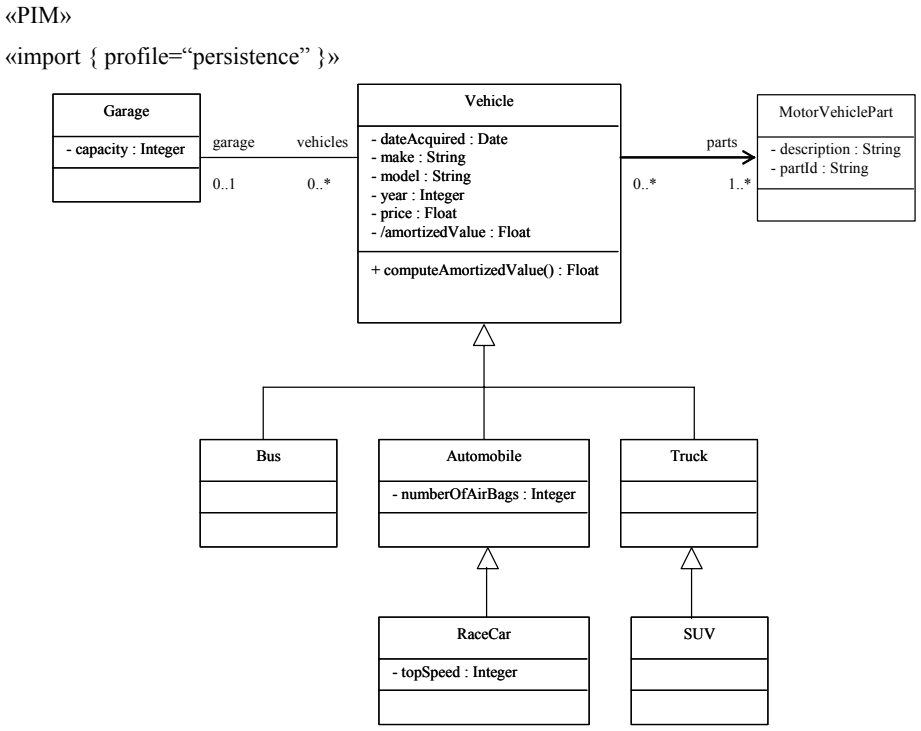


Figure 3 - 2 The Vehicle class diagram

The developer marks the class diagram of the vehicle model from Figure 3 - 1 with «PIM» to indicate that the object model is a platform independence model. Next he imports a persistence profile by putting an «import» and specifying the persistence in the attribute. After the profile is imported, he can use the persistence markings and APIs.

Step 2 – Mark persistence classes with «persistence» and specify identifier(s).

We want to indicate that instances of Vehicle class hierarchy are persistence objects. So we mark them with «persistence». Since Vehicle is the root node of the hierarchy, all subclasses inherit it. Since the Vehicle class does not have an attribute with the stereotype «id», the transformer inserts a new key attribute *id : String* and marks it with «id». We follow the same step for Garage. The *MotorVehiclePart* contains a unique *partId* so we mark this field with «id».

Mercator activates the default PIM persistence transformer when it finds the «import {profile="persistence"}». The transformer then iterates over all persistence classes and locates each non-transient class attribute and association to determine whether their types support persistence. Classes without «persistence» or primitive types are value objects and their contents, not their identities, are stored. If the reachable user-define type of the fields are not marked with «persistence», their instances will not be stored. Therefore it is important to specify all persistence classes whose instances need to be stored.

Now the transformer will create the public *_create(initialId : String) : Object* and *_load (id : String) : Object* and *_findAll(includeSubTypes : Boolean) : Collection* static methods and the public *_getId() : String, _store() : void* and *_remove() : void* methods. The result is shown in Figure 3 - 3.

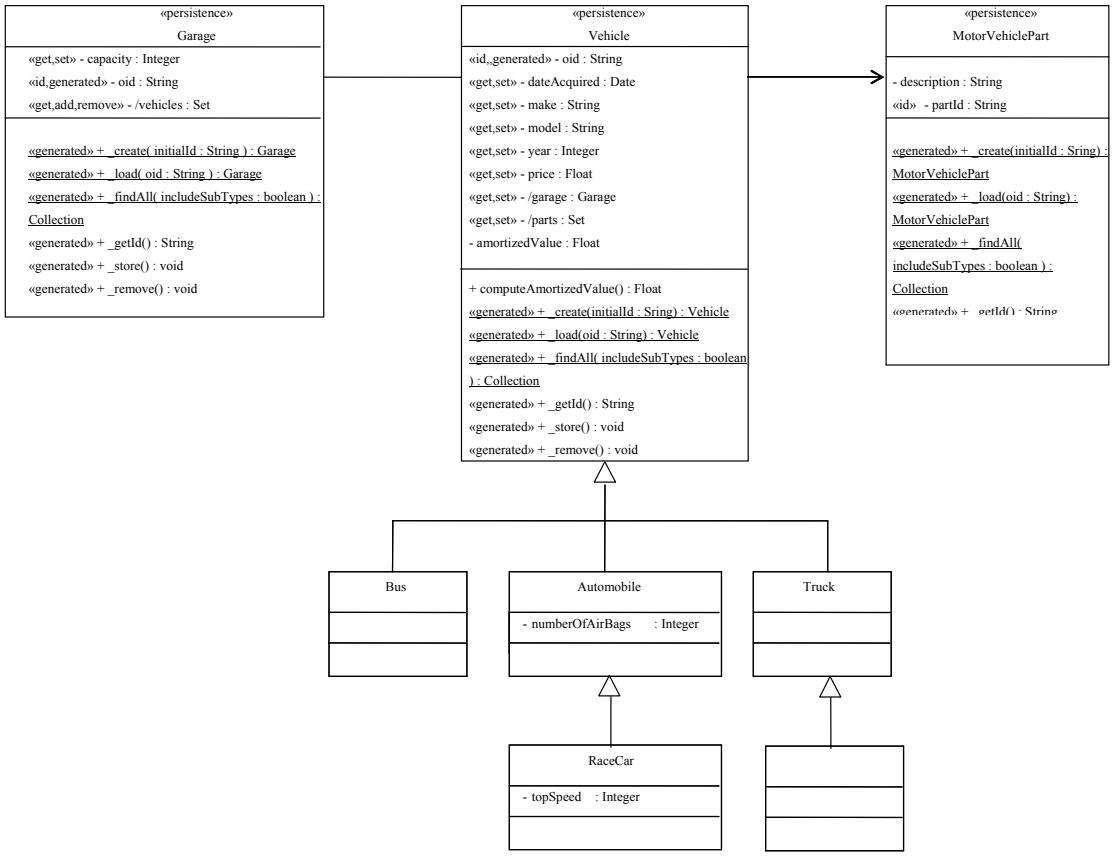


Figure 3 - 3 Marked Vehicle class diagram

Step 3 – Specify the persistence choice.

There are several PSM persistence transformers. For Java-to-XML mapping, we specify the transformation choice as a *javaxml*. The framework looks up the JavaXMLTransformer class from the name of the transformation choice from the transformation registry and delegates the persistence transformation to the JavaXMLTransformer object.

```
   <annotation:annotation xmlns:annotation="http://mercator.org/annotation"
file="vehicle.persistence.annotation.xml">
     <annotation:transformerChoice forNode="*" forModel="psm"
stereotype="persistence::persistence" transformerName="javaxml" />
   …
   </annotation:annotation>
```

Listing 3 - 6 Vehicle persistence annotation file for Java XML

The JavaXMLTransformer uses Java's *XMLEncoder* to implement persistence. The *XMLEncoder* is a Java 1.4 API that uses Java serialization and produces XML output of the object graph from *PersistentDelegate*s. The requirement is that each persistence object must follow JavaBeans specification so that Java can introspect to determine object fields. The mapper iterates over each non-transient attribute and produces its getter and setter methods so as to expose these attributes as JavaBeans.

The JavaXMLTransformer creates a PSM that stores objects into files, each file contains a serialized representation of the instance and the file name contains a unique instance identifier for lookup. The format of the file name is the name of the class followed by '.' and the identifier obtained from *_getId(),* followed by the file extension '.xml'. For example, a Vehicle object with *id* = "1532" will store into a file name 'Vehicle.1532.xml'. This way, each object will be stored in a unique file.

In summary, the JavaXMLTransformer performs the following steps:

- Copy all classes from the expanded PIM to a newly created PSM.
- For each «persistence» class in the PSM starting from the highest level superclass:
    - For each non-transient attribute and association, generate JavaBeans getter and setter.
    - Apply the mapping template to create method body for *_create(), _load(), _ findAll(), _getId(), _store()* and *_remove()*.

After applying the mapping rules, the result of the PIM-to-PSM mapping is shown in Figure 3 - 4.

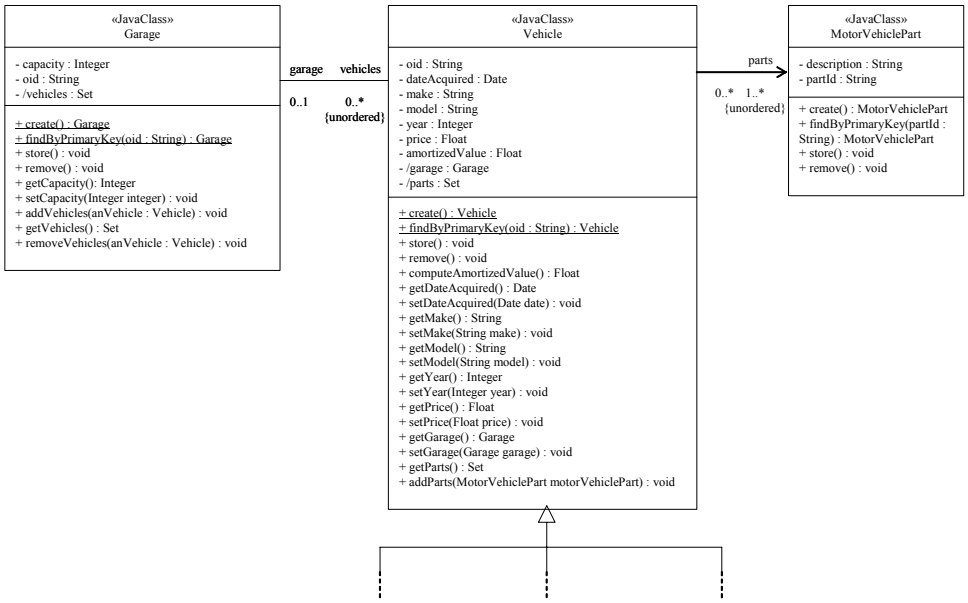


Figure 3 - 4 Java XML PSM of the Vehicle

Step 4 – Customize PIM with annotation parameters. Listing 3 - 7 below shows a default Java XML annotation template file. This file is copied into vehicle.javaxml.annotation.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotation:javaxml xmlns:annotation="http://mercator.org/annotation">
  <annotation:property vendor="SunJDK"/>
  <annotation:property helperClassPackageName="util" description="Package
location for JavaXMLEncoder specific generated classes"/>
  <annotation:property idGenerationStrategy="increment"/> <!-- increment
(default), hilo, UUID -->
  <annotation:property baseDirectory="."/>
  <annotation:property
fileGenerationClass="org.mercator.transformer.XMLFileManager"/>
  <annotation:property transactionSupported="false"/>
  <annotation:property timeoutInSeconds="15"/>
  <annotation:property schemaGeneration="false"/>
</annotation:javaxml>
```

Listing 3 - 7 A default Java XML annotation file

The parameters in the newly created file are customizable and will be used by the JavaXMLTransformer. Parameters such as id generation, based directory of XML persistence files and a Java class name that is responsible for assigning file names to persistent Java objects.

Step 5 – Generate an executable code

The result of JavaXMLTransformer transformation is a Java PSM that can be translated directly into Java code. The Mercator has a default model-to-code transformer that reads a PSM and produces Java source files. The generated code are compiled into Java and executed in a Java runtime environment. Our test cases store persistent objects into files, load them back and compare the object structure (properties and associations). A result of a test case that stores a `RaceCar` object is shown in Listing 3 - 8.

```
<?xml version="1.0" encoding="UTF-8"?>
  <java version="1.4.1_02" class="java.beans.XMLDecoder">
   <object class="vehicle.RaceCar">
    <void property="garage">
     <object class="vehicle.Garage"/>
    </void>
    <void property="numberOfAirBags">
     <int>2</int>
    </void>
    <void property="topSpeed">
     <int>180</int>
    </void>
   </object>
  </java>
```

Listing 3 - 8 A result of persistence file in XML

3.7.2 A CMPEJBTransformer

Step 1 – Create class diagram describing object model

Step 2 – Mark persistent classes with «persistence» and specify identifier(s).

These two steps are independent of any PSM so we follow the same instructions as those in the JavaXMLTransformer example.

Step 3 – Specify the persistence choice.

We choose CMP EJB persistence by specifying the transformer name in the persistence annotation file. The Mercator framework uses this name to look up a transformer, CMPEJBTransformer from the transformation registry.

```
   <annotation:annotation xmlns:annotation="http://mercator.org/annotation"
file="vehicle.persistence.annotation.xml">
     <annotation:transformerChoice forNode="*" forModel="psm"
stereotype="persistence::persistence" transformerName="ejbcmp" />
   …
   </annotation:annotation>
```

Listing 3 - 9 Vehicle persistence annotation file for EJB CMP

The transformer takes the following steps to generate the result:

- Create an empty PSM.
- For each class *A* that contains «persistence»
    - Create Bean interface in the PSM.
        - For local interface, create an interface *LocalA* that extends *javax.ejb.EJBLocalObject*. Add abstract methods from PIM methods. Move business methods' implementation code into the bean class. Mark it with «EJBLocalInteface».
        - For remote interface, create an Interface *A* that extends *javax.ejb.EJBObject*. Add abstract business methods from PIM methods. Each methods can throw java.rmi.RemoteException. Move business methods' implementation code into the bean class. Mark it with «EJBRemoteInteface».
    - Create a bean class, *ABean* that implements javax.ejb.EntityBean. This bean contains bean context and ejb-prefixed methods from the mapping template. Mark this bean class with «EJBEntity».

- Create a bean key class, *AKey* from the «primaryKey» attribute. Implement *equal()* and *hashCode()* methods. Mark this key class with «EJBPrimaryKey».
- Create Home interface.
  - If a *LocalA* is created, add a home interface, *ALocalHome*. Mark it with «EJBHomeInteface».
  - If an *A* is created, add *AHome* that extends *javax.ejb.EJBHome*. Mark it with «EJBHomeInteface».
  - Add *create(), findByPrimaryKey(), findAll, create()*.
  - Tag create() with «EJBCreateMethod».
  - Tag findXXX() with «EJBFinderMethod».
- Put dependency lines from *AHome* and *ALocalHome* to *ABean*. Mark the lines with «EJBRealize».

- Create a deployment descriptor container [Edoc04].
  - If the container does not exist, create a new container.
  - For each EJB, add <ejb> entry in the container.
  - Generate CMR fields from object relationships in the class diagram.

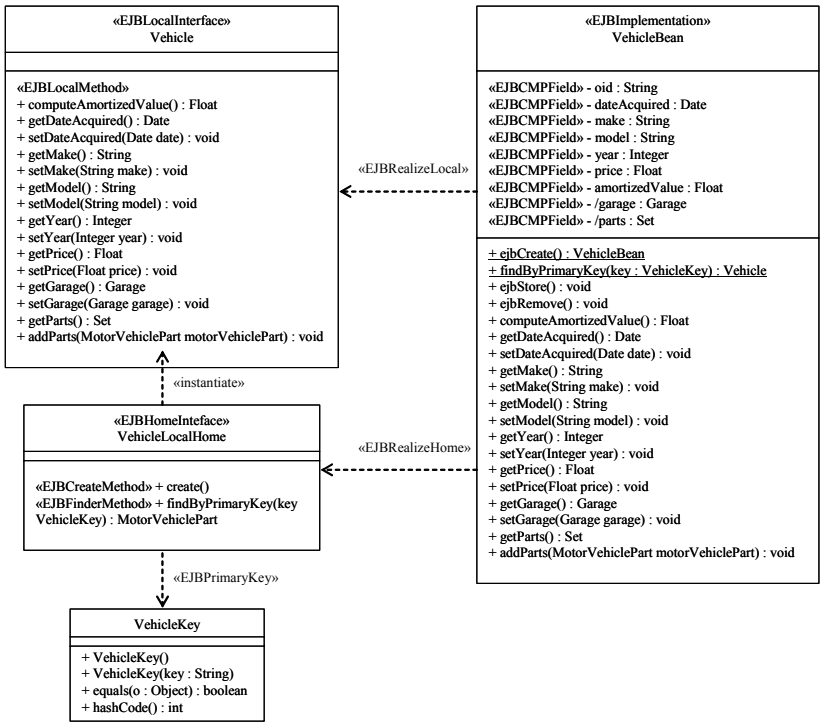


Figure 3 - 5 A partial CMP EJB for Vehicle bean

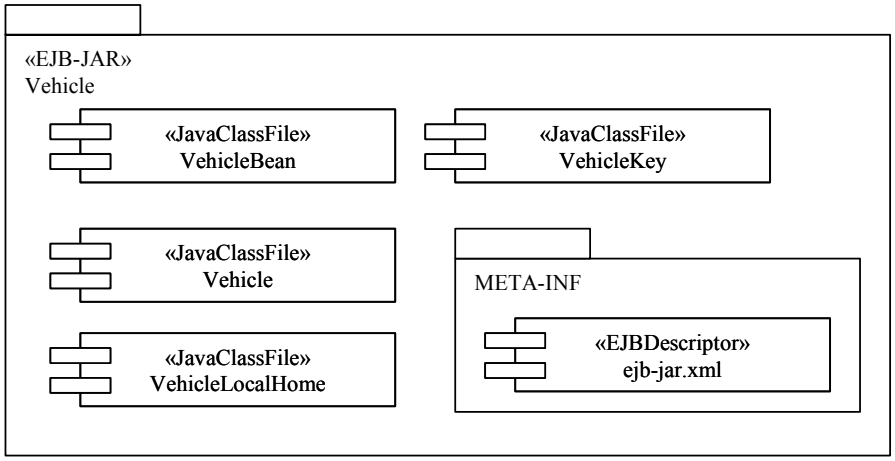


Figure 3 - 6 A partial EJB Implementation model

Step 4 – Customize the PSM with annotation parameters. Listing 3 - 10 below shows a default EJBCMP annotation template file. This file is copied into vehicle.ejbcmp.annotation.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<annotation:ejbcmp xmlns:annotation="http://mercator.org/annotation">
  <annotation:property vendor="SunAppServer"/>
  <annotation:property helperClassPackageName="util" description="Package
location for EJB specific generated classes"/>
  <annotation:property idGenerationStrategy="increment"/> <!-- increment
(default), hilo, UUID -->
  <annotation:property connection.url="jdbc:hsqldb:test"/>
  <annotation:property connection.driver="org.hsqldb.jdbcDriver"/>
  <annotation:property username="sa"/>
  <annotation:property password=""/>
  <annotation:property dialect="HSQLDialect"/>
  <annotation:property transaction.factory="JTATransactionFactory"/>
  <annotation:property transaction.name="java:comp/UserTransaction"/>
  <annotation:property timeoutInSeconds="15"/>
  <annotation:property schemaGeneration="true"/>
   …
</annotation:ejbcmp>
```

Listing 3 - 10 Default EJB CMP annotation file

The parameters in the newly created file are customizable and will be used by the transformer during the PIM-to-PSM transformation. These parameters are:

1. EJB container vendor
2. Datastore provider. In case of relational database, if data tables do not exist, a Data Definition Language (DDL) is created from a table mapping method. Possible mapping methods include top-down, meet-in-the-middle and bottom-up. If data tables already exist, an object-to-table mapping is required.
3. Vendor specific extensions such as session timeout, bean and data cache settings, local transaction settings, locale invocation selection that is not in the EJB specification.

Step 5 – Generate an executable code

The following are steps to generate and optional deploy the result into an executable code.

1. Package all classes into a jar file

2. Generate EJB deployment descriptor, ejb-jar.xml from the deployment descriptor component.
3. Package the jar file, its supported jars and ejb-jar.xml into an enterprise archive (ear) file.
4. Optionally deploy the ear file into an EJB container. Some EJB container supports hot deploy and automatically detect and deploy the new ear. Other container needs manual update or requires a restart.

3.8 Summary

Business applications use a persistence service to store business data. This chapter introduces a persistence profile that modelers can use to specify persistence in their PIMs and shows how they are translated into two concrete implementations; Java XML and EJB container-managed persistence [WJ03]. The persistence profile consists of 10 persistence related stereotypes. Even though each stereotype can have its own transformer, we only created two transformers; one for persistence stereotype and the other for id stereotype. Other stereotypes are markers that these two transformers use to locate persistence methods. They do not have their own transformers. Some profiles only have a couple of stereotypes but their transformers can be very complicated. For example, the Transaction Service profile in chapter 6 only has one stereotype. However, it is complicated because the transaction service depends on the persistence service to indicate which persistence objects involved in a transactional unit of work.

Modelers can choose a persistence method in the persistence annotation file. The prototype currently has a limitation that it supports only one method per profile. Modelers can choose to store data into XML files or EJB CMP but not both. There are a few remedies. We can define specialized profiles; one for each concrete persistence method. However, this approach makes the profiles depend on middleware. Alternatively, we can specify a global persistence method and override the persistence method to specific class elements. Listing 3 - 6 and Listing 3 - 9 show an annotation attribute, *forNode,* in the transformation choice that can be used to indicate the scope of the transformation. By default, the value '*' indicates a global transformation (see

section 8.4). We can specify specific nodes to use different transformation methods. Nodes can be classes or packages. If the node is a package, it means that all classes within the package will be transformed by the same transformer.

Objects need to have a unique identifier. Persistence id transformer generates code that assigned an id to each persistent object. Alternatively, the identity assignment can use the naming service (chapter 4).

Persistence is the central service that others depend on. Transaction service (chapter 6) use persistence service to identify persistence objects that participate in a unit of work. Distribution (chapter 5) and messaging (chapter 7) services use persistence to serialize objects during object transmission. However, they require that the persistence method must produce streamable objects that can be sent over the network. Otherwise, a transformation conflict must be raised. Other transformers check the streamablility property from the *streamable* attribute of the persistence annotation file. The transformation will succeed only when there is no more conflict.

Chapter 4 Naming Service

4.1 Introduction

This chapter proposes a platform independent model for naming service that uniformly manages object names in a single machine or a distributed environment. A naming service associates names with objects (name binding) and provides a facility to look up objects by names (name resolution) in a distributed environment. A naming service can be used to find printers, files, machines, networks, named services or just object references. A name server manages name registration, look up, removal and coordination with other name servers. Most name servers group names into a hierarchical tree-like structure where leaf nodes associate an object reference with its name and non-leaf nodes are directories that have names and their own set of tree nodes. The root of a tree represents a naming context and can be nested into another tree to form a bigger name space. A naming service is a basic facility that is used by others. The persistence service uses it to map persistence objects into files, database or network streams. The distribution service uses it to bind and resolve distributed objects for distributed invocations. The transaction service uses it to define unit-of-work boundary and propagate transaction contexts across machines. Messaging service uses it to publish topics or queues that other objects can subscribe to. Thus, nearly any service uses the naming service instead of implementing this facility by its own.

There are two primary problems with naming. First, there are semantic naming differences between single and distributed execution environment. In a single address space environment, naming is implicit by using memory address bound to a variable. As long as a program holds references to objects and passes them to others, there is no need for explicit naming. In a distributed environment, address space spans more than one machine. Unless it uses a shared memory space, it is not feasible to use memory addresses to uniquely identify objects. Some programming languages do not expose memory addresses of objects. Therefore, programming model for object instantiation,

lookup and destruction will be different. This forces domain models to reflect object location explicitly in the design. Designers must decide early on where each object will reside and how to identify it. Once the decision has been made, it will be difficult to change later, e.g. changing from a local object into a remote object.

Second, there are many different implementations of naming service. Each implementation has its own naming scheme, binding, resolution and management. Even though Java provides a standard JNDI application programming interface, remote object proxies (CORBA stubs, RMI stubs) returned from the lookup need to be cast differently. An attribute-based naming scheme such as X.500 [ITU05] allows for descriptive queries in addition to a typical name-node property and therefore requires different name mapping and lookup strategy.

Names doe not contain physical address information at a PIM level. Object locations in PIM are based on logical node-unique names. A logical node manager is responsible for 1) node-level, unique name assignment, 2) local object binding and 3) remote object resolution. For instance, an account object does not have to know whether its customer information object resides in the same machine or not. A developer defines a PIM and writes code as if they were all in the same address space. Later, the developer describes logical object locations (nodes) separately in a platform independent naming annotation. A node is a logical container that keeps information about objects in its naming context and defines a communication boundary between objects. Local objects are created and used within the same node while remote objects are created in one node and used in others. To lookup objects, a client asks for the object references from the logical node manager.

A platform independent naming service transformer uses the naming annotation to identify classes whose instances are created and invoked across node boundary. The developer specifies a node-to-machine mapping strategy in a platform specific naming annotation. A logical node can be mapped into a cluster of physical machines or a machine can be assigned to more than one logical nodes. For each node, the developer

indicates which concrete naming service is used. A platform specific naming transformer generates an implementation based on the naming service choice and its naming annotations. Since naming in PIM is logical and independent of naming methods, the model can be easily reused or changed. The developer simply changes object locations in the naming annotation file and retransforms the same untouched PIM into a different implementation. Another benefit of separating naming choices from the PIM is the support of multiple naming services. The designer can choose more than one concrete naming services of the same node and the transformer will automatically generate multiple implementations of object binding and resolution of the same object to different name servers.

4.2 A Motivating Example

Consider a simple case. A *Client* object creates an instance of class *Loan*.

```
// class Client
public static void main(String args[]) {
    Loan loan = new Loan();
    loan.approve(…);
}
```

Listing 4 - 1 A simple object creation statement

This statement contains two expressions; the object instantiation (*new Loan()*) and the object reference assignment (*loan = new Loan()*). If these two objects execute in the same address space, the code will work just fine. But if one of them executes in another address space, the developer has to change a lot of code. He has to decide whether the remote object instantiation should be at a server or the client and how to return the remote object reference from the server back to the client. Suppose the object instantiation is executed at the server, one possible implementation using an RMI might be:

At the server side:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface Loan extends Remote {
    // some remote method
}

import java.rmi.*;
import java.rmi.server.*;

public class LoanImpl extends UnicastRemoteObject
                          implements Loan
{
    public LoanImpl() throws RemoteException {
        super();
    }

   // some remote method implementation

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "rmi://localhost:1099/Loan";
        try {
            Loan loan = new LoanImpl();
            Naming.rebind(name, loan);
        } catch (Exception e) {
            System.err.println("Loan exception: " +
                        e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Listing 4 - 2 A loan implementation at the server using RMI

Since the Loan object is created at a server machine, it needs to bind itself with the server RMI registry. The caller obtains the remote object reference from the loan object by looking up the bound name of the object from the server naming server:

```
import java.rmi.*;
import java.math.*;

public class Client {
```

```
        public static void main(String args[]) {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            try {
              String name = "rmi://" + args[0] + "/Loan";
              Loan loan = (Loan) Naming.lookup(name);
              // call some method
              loan.approve(…);
            } catch (Exception e) {
                System.err.println("Exception: " +
                  e.getMessage());
                e.printStackTrace();
            }
```

Listing 4 - 3 A loan lookup at the client using RMI

The client must know the location of the server in addition to the name of the remote object.

If the program is migrated to the CORBA naming service, there are many changes required. The server must create an object request broker, obtain and activate the root object adapter, bind the loan object and start the object broker process.

```
   public class LoanImpl extends LoanPOA
                              implements Loan
   {
        public LoanImpl() throws RemoteException {
            super();
        }

       // some remote method implementation

        public static void main(String[] args) {
            String name = "rmi://localhost:1099/Loan";
            try {
              // init ORB
              ORB orb = ORB.init(args, null);
              POA poa =
POAHelper.narrow(orb.resolve_initial_reference(“RootPOA”));
              posa.the_POAManager().activate();
```

```
            Loan loan = new LoanImpl();

            org.omg.CORBA.Object obj = poa.servant_to_reference(loan);
            System.out.println(orb.object_to_string(obj));
            orb.run();
        } catch (Exception e) {
            System.err.println("Loan exception: " +
                        e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Listing 4 - 4 A loan implementation at the server using CORBA

The object is bound to a CORBA IOR string that is printed out (or saved into a file) and the client uses the string as an input parameter to lookup the object reference.

```
public class Client {
    public static void main(String args[]) {
        try {
          String name = args[0];
          ORB orb = ORB.init(args, null);
          org.omg.CORBA.Object object = orb.string_to_object(name);
          Loan loan = LoanHelper.narrow(obj);
          if (loan == null) throws …
          // call some method
          loan.approve();
        } catch (Exception e) {
            System.err.println("Exception: " +
              e.getMessage());
            e.printStackTrace();
        }
     }
}
```

Listing 4 - 5 A loan lookup at the client using CORBA

The above example assumes that the Loan object at the server must be created and bound to a name server (RMI registry or CORBA ORB) before the client can obtain the remote

object reference. If the client wants to take control of the remote Loan object instantiation, one possible implementation is using a factory method.

```
LoanFactory loanFactory = ….
Loan loan = loanFactory.create();
loan.approve(…);
```

Listing 4 - 6 A loan object is created by a factory object

Now the problem shifts to how to obtain a remote factory object. If there are many remote classes, should we create a remote factory for each one or is it better to have one object responsible for remote object creation, lookup and destruction? How do we assign names to them so that we can look them up later? Is it possible to share the name to a pool of objects so that an arbitrary instance in the loan pool can be returned? To abstract these details in a PIM, a middleware independent naming service must address these questions.

Another use case is when we need to assign a unique identifier to each object but we don't want to name each object ourselves. We want the system to generate a unique name that can be used to identify the object later. We want the flexibility to choose a name generation strategy as simple as a counter or more complicated as UUID. This object identifier can be used for object persistence. Therefore, a middleware independent naming service must provide the system generated naming assignment and guarantee the uniqueness of the name in a given node. The design of the middleware independent naming service must address naming characteristics described in the next section.

4.3 Naming Characteristics

Sinha [Sin96] shows the characteristics of a good naming service that shape the design of a platform independent naming service.

4.3.1 Location transparency

The name of an object should not contain the physical location of the object container. PIM should use logical object location. The middleware independent naming service

must separate logical object and node names from physical revealing object and machine locations. It must provide standard APIs for transformer developers to map between them. Model developers work with logical names and do not have to be aware of actual naming services used.

4.3.2 Location independency

To accommodate object migration, the name of the object should not change even when the object's physical location changes. The location independency improves scalability and robustness of the distributed system. Even though object migration is beyond the scope of this dissertation, the use of logical object and node names decouple object physical locations and provide a basis for object replication and migration [Lam86].

4.3.3 Uniform and meaningful naming convention

A name is a textual unit by which a thing is known. Names are defined by a naming scheme. However, there are many different naming schemes that are used by different naming middleware. Among them are:

- Memory address
- Directory based file system
- URN/URL/DNS/CORBA corbaloc
- Attribute based X.500, e.g., { o=UIUC, ou=cs, cn=Student1 } where o is an organization name, ou is an organization unit, cn is a common name.
- Novell Directory System (NDS)
- UUCP e.g. host-1!host-2!host-3!witthawa
- Binary encoded name such as CORBA IOR protocol

CORBA IOR uses a binary format for names while most others such as Java RMI, URN, X.500, CORBA corbaloc use a textual format. X.500 uses attribute based naming while most others use hierarchical directory structure. Middleware independent naming service must use a logical hierarchical based naming scheme. Name transformers are responsible for mapping logical names into concrete naming specific schemes.

4.3.4 Multiple user-defined names for the same object

Even though a logical name must be unique, it may be mapped into names from different naming schemes. It is possible to resolve the same object from different name servers. More specifically, different proxies of the same named object can be resolved from clients that use different concrete naming services. Concrete naming servers can be an RMI registry, LDAP server, JNDI service provider, web service request listener or implicitly a java virtual machine that manages its object references in its virtual address space.

On the other hand, a name can link to more than one instance of the same class. This is commonly used in an object pool pattern where an instance in the pool can be looked up, used by a client and returned to the pool when finished. For example, common pools are servlets and database connections. Using pooled objects improve performance and scalability.

4.3.5 Performance

One of the reasons distributed systems have a different design than a system on a single is for performance. They want to be able to control the object life cycle and the way objects interact to improve network efficiency. If an object is created and used within a code block, there should not use naming service. If the object is created and looked up later from another object that do not have references to the object, the object and its name must be stored in a map-like data structure or cache and be accessible from a local name server. If the object is looked up from another object in a different machine, it must be bound and resolved from a global naming server. Mercator naming transformer must keep track of object creations, references and invocations in order to identify which objects are local or remote. While a naming service is responsible for object binding and resolution, object invocation will be further analyzed in the distribution service chapter.

4.4 Platform Independent Naming Profile

The Naming profile consists of 8 stereotypes and 1 library class. A developer marks classes whose instances need names with «name». The developer specifies the logical

name in the stereotype attribute. A «singleton» class is a named class that has at most one instance while a «pool» class is a named class whose instances are at most as indicated in the *size* attribute. These stereotypes contain a lazyCreation attribute. If it is true, an instance is created when a program executes a 'new' operator. If it is false, instances will be pre-initialized. A package or a model reside in a logical node and is put with «node» with the *name* attribute. Every time a package or a model contains «node», all containing classes can refer to a NodeManager object.

A Naming class is a singleton that manages object life cycle inside a node and communicates with other naming singletons in different machines. It intercepts call to 'new' operator so that it can keep track with the number of instances of the class and binds the object with a name if the object is a newly created remote object. When Mercator transforms the code in PIM and detect that the caller of the new statement does not run in the same node as the to-be-created object, it will convert all 'new' statements from:

```
ClassA a = new ClassA();
```
to:
```
ClassA a = (ClassA)Naming.create(ClassA.class);
```

However, intercepting object destruction is tricky since Java has no explicit statement to destroy object. Naming transformer implicitly inserts Naming.destroy(a); at the end of the code block to do a housekeeping work to update the number of instances of the class of the destroying instance or deregister the object if it is a remote object.

The Naming class also provides a static helper method *lookup()* that returns an instance of the class with the logical name. The transformer generates the *getName()* code in the PIM to return a globally unique logical name from the input object. Since PIM uses logical names. Their fully qualified global names are the concatentations of the node name and the logical names. A PSM name transformer generates a *Naming* implementation to map the logical name into a physical name appropriate to concrete

naming choice. For example, a logical qualified name node1::ClassA1 is mapped into rmi://localhost:1099/ClassA1 for a RMI name server or corbaloc:iiop:localhost:4444/NameService/ClassA1 for CORBA name server or http://localhost/ClassA1 for a web service. The PSM naming transformer must implement *getName()* and *getQualifiedName()* in the PSM to return a physical name such as ClassA1 and a fully qualified physical name of the object such as rmi://localhost:1099/ClassA1 for RMI naming service choice.

All Naming methods throw unchecked exceptions if there are errors during object binding or resolution. For example, an *InvalidNumberOfInstancesException* if the *create()* finds that the newly created object exceeds the number of instance limit for the class.

The NodeContainer class defines a node boundary and provides a name service context for callers. A node container cannot map into more than one physical machine but a physical machine may contain more than one node container which each of them run in a separate process.

A model uses naming service by importing the naming service profile. Once imported, the name service annotation file is read (or created with default values if it does not exist). Mercator associates this annotation the PIM and allows the developer to annotate the model and keep the annotation separate from the model.

Stereotype	*Applies to*	*Attributes*	*Description*
«import»	Model	{profile=”Naming”}	Indicate that contained model elements may use naming service and attach a naming annotation to the model.
«name»	Class	{name: String,	A class whose

		lazyCreation : Boolean}	instances have names.
«autoName»	Element	{ namePolicy = "counter", "UUID", lazyCreation : Boolean }	A named element whose name is automatically generated.
«singleton»	Class	{name: String, lazyCreation : Boolean }	There is at most one instance of this class.
«pool»	Class	{name: String, size : Integer, lazyCreation : Boolean }	There are at most *size* instances of this class.
«node»	Package, Model	{name : String}	Logical node name.
«create»	Operation		Indicates an operation that creates an object. Omitted if the operation name is create() or the keyword *new* is used.
«lookup»	Operation		Indicates an operation that finds an object. Omitted if the operation name is lookup().
«destroy»	Operation		Indicates an operation that destroys an object. Omitted if the operation name is destroy().

Table 4 - 1 Naming Stereotypes

4.4.1 Platform Independent Naming APIs

PIM uses platform independent APIs and does not have to know how the APIs are implemented. For example, the static operation of the class, _create(), may delegate to a factory, a builder, or a name server. The choice of the concrete implementation depends on which naming choice is chosen and the transformer implementation of that choice.

The «node» stereotype indicates a logical execution node of running instances and a tag value *name* indicates a logical node name. A Class can have its instances running in more than one node. If a developer applies the stereotype to a Package or a Model element, all instances from the contained classes in the package or model will run on that node. The developer can apply different node names to specific classes than those in containing Package or Model to indicate that their instances will run in different nodes. This information will be used to determine if a calling object should make a remote invocation to a called object. If they reside in the same node, the invocation will be local. However, if each object can contain in different nodes, the invocation must be remote.

Metamodel element	*Inherited methods*	*Stereotype if method name is different than default*	*Description*
Class	+ _getName() : String	«name»	Returns a node-level unique name of the object.
	+ _create(logicalName:String) : Object	«create»	Create and assign a name to an object.
	+ _lookup(logicalName: String) : Object	«lookup»	Resolve an object from a name
	+ _destroy()	«destroy»	Destroy an object and remove it

			from the naming service.
NodeContainer	+ _getNodeName()	«node»	Returns a logical node name.
	+ _getNameService() : Naming	«nameService»	Returns a container's naming service instance.

Table 4 - 2 PIM Naming APIs

After a PIM imports the Naming profile and applies the «naming» to its model, it can use the APIs defined in the above table. The implementation of these methods and the NodeContainer class depends on a naming transformation choice. For example, if the developer chooses a CORBA IOR naming scheme, the implementation would look like the example in Listing 4 - 5. The developer can customize object names by overriding the *_getName()* but it must make sure that the name is unique within its node container.

4.4.2 Logical Naming Scheme and Resolution

In a single address space, an object is usually identified by its memory address. A variable contains the object reference and can be passed between methods. In this case, the variable is a name binding that associates a memory address with the object. Therefore we do not need explicit name binding. Only when a program does not have references to objects, it will need a way to look up the objects probably from a lookup table, an object cache, a database or an external service. That way, it needs a mechanism to identify and obtain object references. It is also the case for a distributed environment since a memory address pointed by an object reference in one machine is not unique and cannot be used to identify an object. In addition, we do not store and retrieve actual objects that reside in another machine. Instead we retrieve object stubs that act as communication bridges between caller objects and the remote objects and invoke operations through the stubs. There are many ways to identify objects. One way is to use a machine name plus an object reference to form a global object identifier. An object

reference can be a memory address, a class-level counter or a unique string assigned by an object manager. However, we want to achieve the location transparency and meaningful naming convention in a PIM, we will use a logical node name instead of a machine name and choose a textual object name over a binary object reference. The logical node is a developer assigned name in a platform independent naming annotation. Be default, all instances are created and run in a default logical node named 'Node1'. If the developer creates a new node and put classes into it, all instances from this class will be created in that node. It is possible to put classes into more than one node but the developer must describe which instances are created in which nodes. The textual instance name is obtained from an instance method _getName().

During the mapping process, the developer chooses a name server mapping choice to the input PIM. Mercator looks up a transformer based on the choice and uses it to map logical object identifiers in the PIM to physical object names. The physical object name contains two parts; the physical node and the object identifier. The physical node is a computer name and a containing process of the running instances, or a computer name plus a socket port number that is created by the name server inside the containing process. Since these two information will not be known until runtime, a special class NodeContainer has two static methods, _getNodeName() and _getNameService(). The transformer creates a code that calls these two methods to obtain the physical node. The transformer obtains the object identifier from the class method _getName().

If an object is created and used by others in the same node, the transformer will not change anything. On the other hand, if the object is created by another object in another node, the transformer will 1) generate a name server bootstrap in the called node if it is not created, 2) create the object, 3) obtain a unique instance name from the object by calling _getName(), 4) register the name and the instance to the name server. If the object is accessed from another object in another machine, the transformer will 1) obtain the physical machine from the logical machine mapping, 2) connect to a name server running on the physical machine, 3) map the logical name to a physical name and look up the object stub, 4) return the object stub to the caller.

There are 3 types of object multiplicities that constraints object instantiation; a singleton, an object pool and a regular object. A singleton contains only one instance per class and the name will always be the same. An object pool contains a limited number of objects and may return duplicated names if objects are recycle. A regular object does not have limits on the number of instances. A transformer may use a static method to keep track of the number of objects and check whether a new instance creation is allowed or not or it can delegate this task to a factory object. The object creation strategy can be lazy initialized, pre-initialized. The lazy initialized strategy creates objects when they are needed while the pre-initialized strategy creates a singleton or object pool ahead during the name service initialization. Object life cycle can be client dependent where clients must explicitly destroy objects or leasing where objects have expiration period associated and will be destroyed as needed by a name service if they are not used within the expiration period.

4.4.3 Naming Annotation

The Mercator name transformer uses a naming annotation file to store model naming configuration. This annotation file is created when a model imports the naming profile. If the file already exists, Mercator reads the file and associates the annotation to the PIM. By default, all instances reside in a default logical node *Node1*.

```
<pim:annotation name="Naming">
  <node name="Node1">
    <instances>*</instances>
  </node>
</pim:annotation>
```

Listing 4 - 7 Default naming annotation

The wildcard (*) indicates that all instances from all classes in the PIM are created in a logical node *Node1*. To define objects in a different node, we must create a new node definition and indicate which classes whose instances execute in the new node. It is possible that one class has instances running in many nodes. Below is an example of a model that instances from a class *Loan* in a *org::mercator::test* package reside in *Node2* while the rest reside in *Node1*.

```
<pim:annotation name=”Naming”>
  <node name=”Node1”>
    <instances>*</instances>
  </node>
  <node name=”Node2”>
    <instances>org::mercator::test::Loan</instances>
  </node>
</pim:annotation>
```

Listing 4 - 8 Defining the second logical node

Naming annotation also supports wildcard at a package level. For example, if all instances of classes defined in a package *org::mercator::test* reside in *Node2*, it can be defined as:

PIM annotation:

```
<pim:annotation name=”Naming”>
  <node name=”Node1”>
    <instances>*</instances>
  </node>
  <node name=”Node2”>
    <instances>org::mercator::test::*</instances>
  </node>
</pim:annotation>
```

Listing 4 - 9 All instances from the org.mercator.test package belong to Node2

Mercator uses these node locations to identify whether object creation and invocation should be done locally or remotely. Naming annotation in PSM specifies where nodes are mapped into physical machines.

PSM annotation:

```
<psm:annotation name=”Naming”>
  <mappings>
    <mapping from=”Node1” to=”127.0.0.1” />
    <mapping from=”Node2” to=”mercator.cs.uiuc.edu” />
  </mappings>
</psm:annotation>
```

Listing 4 - 10 A PSM naming annotation define logical to physical name mapping

4.5 PIM-to-PSM Transformation

Consider a simple case. A *Client* object creates an instance of class *Loan*.

PIM:

```
// class Client
Loan loan = new Loan();
```

Listing 4 - 11 A simple Loan creation statement

The Mercator's PIM transformer parses and generates the PIM AST node as follows:

```
<Statement source="Loan loan = new Loan();">
  <CreateVariable name="loan" type="Loan"/>
  <AssignAction>
    <From>
      <CreateInstance classifier="Loan"/>
    </From>
    <To>
      <Variable name="loan"/>
    </To>
  </AssignAction>
</Statement>
```

Listing 4 - 12 An AST representation of the creation statement

When the developer imports the naming profile, Mercator looks up the profile definition file (naming.profile.xml) and checks whether this model has a naming annotation file. if there is not, Mercator creates a default naming annotation with one logical node, *Node1*.

```
<annotation name="Naming">
  <node name="Node1">
    <instances>*</instances>
  </node>
</annotation>
```

Listing 4 - 13 Logical node definition

If the Naming profile is not imported into the PIM, the PIM Transformer does not touch the code and all code generation is created normally. However, when the Naming profile is imported and the «naming» stereotype is applied to the PIM, the

PIMNamingTransformer that is associated with the stereotype will be triggered when the model is transformed. The transformer checks the naming annotation and keeps track of object locations. When the transformer analyzes the statement in Listing 4 - 11, it checks the node location of the *Client* and *Loan* classes. If they are different, the transformer will set the *nameServiceRequired* flag on the model. Later, the PSM's *postTransformer()* will check this flag and generate naming service bootstrap code based on the concrete naming service choice. Next, the PIMNamingTransformer changes the input PIM code into:

Intermediate PIM:

```
Loan loan = Naming.create(Loan.class);
```

The name transformer must keep track of all created objects so that it can verify that object creation satisfied the total number of instance constraint imposed by the «singleton» and «pool» stereotypes. Since there are many ways to create an object in Java, the transformer converts the name creation code so that it delegates the name creation to the Naming class. For example,

```
Loan loan = new Loan();
```

is semantically equivalent to:

```
Loan loan = Loan.newInstance();
```

The Mercator's name transformer changes either of them into:

```
Loan loan = Naming.create(Loan.class);
```

If the loan object is defined as a singleton but is created twice in the same scope, the transformer will throw an exception.

4.5.1 Naming Server Bootstrap

PIM uses *Naming*'s static methods to create, lookup and destroy objects. These static methods are implemented by the PSM by delegating their tasks to a concrete naming service object. If the *nameServiceRequired* is set, a naming service bootstrap code must be generated. The name service object is unique within a node and therefore for each logical node, a name service bootstrap will be generated. It is possible to have more than one logical node running in the same physical machine. Consider the bootstrap for RMI and CORBA implementations for the *Naming* class.

PIM:

No bootstrap code required.

RMI PSM:

```
public class Naming {
    static {
    RMISecurityManager security = new RMISecurityManager();
    System.setSecurityManager(security);
    // start a registry daemon
    LocateRegistry.createRegistry(getNameServicePort());
    }
}
```

CORBA PSM:

```
public class Naming {
    private static NamingContext ctx;

    static {
    Properties props = new Properties();
    props.put("org.omg.CORBA.ORBInitialHost", NodeConatiner.getNodeName());
    props.put("org.omg.CORBA.ORBInitialPort", "" +
getPhyscialNameServicePort());
    ORB orb = ORB.init(null, props);
    POA poa = POAHelper.narrow(orb.resolve_initial_reference("RootPOA"));
    poa.the_POAManager().activate();
    NamingContext ctx =
NamingContextHelper.narrow(orb.resolve_initial_references("NameServer"));
    }
}
```

Listing 4 - 14 RMI and CORBA bootstrap implementations

4.5.2 Instance name mapping and lookup

A PIM uses logical node and object names to identify objects while PSM uses concrete names based on naming service choices and the node mapping in the naming annotation. PSM Naming transformers map these logical names into concrete names by implementing the static method Naming#_mapLogicalToPhysical(). The Naming#lookup() calls this method to obtain the concrete name of the object. Table 4 - 3 hows concrete names of a logical instance loan1 of a class org::Mercator::test::Loan.

Concrete naming service	*Mapped name*
RMI	rmi://127.0.0.1:1099/org.mercator.test.Loan/loan1
JNDI	java:comp/env/ejb/org.mercator.test.Loan/loan1
CORBA	iiop://127.0.0.1:1500/ogr.mercator.test.Loan/loan1
LDAP	ldap://localhost:389/loan1

Table 4 - 3 Name mapping for Nod1/org::Mercator::test::Loan/loan1

Transformer developers can create custom naming transformers to map logical node names to other naming schemes by generating different code in this static method.

4.5.3 Remote instance creation and binding

The *Naming* class encapsulates life cycle management of objects. If an object resides in its node, the *Naming* object will instantiate the object, bind it with its node name and return the new object to the caller. However, if the object resides in different nodes, the *Naming* will contact the remote Naming and delegate of the instance creation to the remote name server. The Naming receives the result and returns it to the caller. The result can be either a remote proxy if the object is a persistent object (pass-by-reference) or a value (pass-by-value). The detail of object passing semantics is described in the distribution service chapter.

Once the *Naming* object creates an instance, it will bind the instance with a name obtained from its object's _getName(). If a client looks up the object from the name and

the name is already bound, the bound object is returned. Otherwise, an unchecked *InstanceNotFoundException* is thrown. The instance of the Naming class is a name server that must be created before a naming service can be used.

PIM:

```
// client @node {name=”node1”}
Loan a = Loan._lookup(“loan1”);

@node {name=”node2”}
public class Loan {
…
}

@generated
package org.mercator.test;
public class Loan {
  public Loan _lookup(String logicalName) {
    return (Loan)Naming._lookup(logicalName, Loan.class);
  }
}
```

PSM:

```
public class Naming  {
  public static Object _lookup(String instanceName, Class cls) {
    String concreteName = cls._mapLogicalToPhysical(instanceName);
    return cls._lookupPhysical(concreteName);
  }
}
```

RMI PSM:

```
public class Loan {
  public static Loan _lookupPhysical(String concreteName) {
    return (Loan)java.rmi.Naming.lookup(concreteName);
  }
}
```

CORBA PSM:

```
public class Loan {
  public static Loan _lookupPhysical(String concreteName) {
    Object obj = Naming.lookup(concreteName);
    Loan result = (Loan)PortableRemoteObject.narrow(obj, Loan.class);
    return result;

}
```

Listing 4 - 15 Lookup method implementations using RMI and CORBA

4.5.4 Object destruction

The *Naming* object is responsible for destroying and unbinding objects no longer used. Since Java does not require explicit object destruction, the transformer inserts object._destroy() statement at the end of code block unless the object is a value returned by the current code block. If this is a case, the object destruction code is put into its parent code block.

4.5.5 Performance

Even though we can hide object registration and lookup from a PIM, it does not mean that we should freely assign object locations. Object binding and lookup incur overhead. Good design discourages fine-grained object communication [Fow03]. Instead, the Naming service should be used for coarse-grained object lookup such as a Remote Façade [Fow03] or Service Locator [AMC03].

4.5.6 Integration with External Naming Systems

The naming profile assumes that class model is built and generated from the Mercator. However, it's also possible to integrate to existing systems that have their own naming services by implementing a new PSM transformer that generates an adapter to map and resolve names between different naming schemes.

4.6 Summary

This chapter introduces a naming service profile that allows PIM to create, look up and destroy objects using logical names and well defined platform independent APIs. The profile contains platform independent APIs, a NodeContainer class, stereotypes and

annotations that developers can use as well as platform specific APIs that transformation developers must implement. A logical name uses a logical node name and a locally unique name assigned by a local name server to identify objects. A client program refers to local and distributed objects by these logical names and does not have to be aware of name servers or actual concrete names. A local name server is responsible to map logical names into implementation specific names and to communicate with other name servers in order to register and resolve remote objects. The naming service is a fundamental service that is used by persistence service in the previous chapter as well as other services we will introduce in next chapters.

Chapter 5 Distribution Service

5.1 Introduction

Object distribution is a key concept in enterprise distributed computing. It allows objects in one environment to invoke remote operations of objects in a different environment. In addition, objects can be passed and returned as parameters during the remote invocation. While developers can write code that manages the distribution concern directly, the task is often done by using services from middleware. Middleware provides distribution transparency[1] by facilitating remote object stubs, naming and lookup service, object graph serialization and life cycle management. However, the use of middleware service is not transparent. Each middleware has its own programming model and enforces the way object models are constructed and behaved. This leaves many middleware specific artifacts in the object models and makes it harder to understand or migrate systems that use one middleware to another. Not only does the dependency to middleware specific artifacts make the object model more complicate, but each middleware also differ on how remote objects are located, passed and invoked.

Enterprise system design must foresee the distribution concern since its beginning which means that designers must choose a particular middleware and use APIs specific to that middleware in their object model. Once the middleware specific artifacts are embedded into the object model, the middleware lock-in is inevitable.

Attempts have been made to create a one-for-all middleware standard but so far there is no clear winner. CORBA is one accepted standard that promotes interoperability among system developed in different programming languages. However, others have been

[1] Distribution transparency includes access, location, concurrency, failure, migration, persistence and transaction transparency [RM-ODP]

introduced and it's unlikely that one middleware will serve all purposes. Instead we should accept middleware coexistence and embrace them.

The purpose of this chapter is to define an abstraction level that allows object modeling without being concerned about distribution and transformation rules that translate object models into ones that use specific middleware. Information specific to each middleware implementation will be stored in the transformation rules as well as the annotation files; one for each middleware. We show that the separation of middleware annotations and the transformation rules allows object models that do not depend on distribution concern; thus making model reuse plausible.

We will illustrate the object distribution implementation with a simple example. Suppose a developer in a bank designed an interest calculation processor that accepted various calculation algorithms and returned an annual interest amount. A simple class model of the system is depicted in Figure 5 - 1.

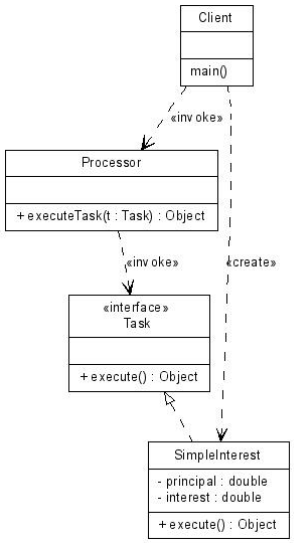


Figure 5 - 1 Simple class model

He defines an interface, *Task,* that contains a generic `execute()` method. A `Processor` class has a `executeTask()` method that takes a *Task* object and delegates the interest calculation to that task and return a value. A `SimpleInterest` class implements the Task interface and returns an interest amount based on a principal and the current annual interest rate obtained from a database. Another client can use a different interest

calculation algorithm by replacing the `SimpleInterest` with another class that implements the `Task` interface. This will allow for more complex interest calculation based on customer account type, current balance level or credit history.

The implementation of this system is trivial and can be done with any object oriented language such as Java. However, there are four problems that may arise when the system is used in a distributed environment.

The first problem is that the class model in Figure 5 - 1 assumes that instances of the classes reside in the same computing space which is fine if the system runs standalone. However, once the system grows, it may be possible that a `Client` object is created and executed in a different machine than the `Processor` object. The `SimpleInterest` object may be instantiated at a client or a server. The configuration of object location will impact on how the system invokes and passes objects to the calling objects. Figure 5 - 2 shows three possible configurations of three participating objects. The first case, all objects reside in the same executing environment so that the object invocation and the object passing are done locally. The second case contains two nodes where the Client instantiates a SimpleInterest object and remotely create and invoke Processor's execute(). The last case has a Client to remotely create both Processor and SimpleInterest that reside in the same machine. Each configuration uses different implementation strategy. The configuration of object partitioning may not be known in advance or they may change due to the system upgrade or from the integration of other systems.

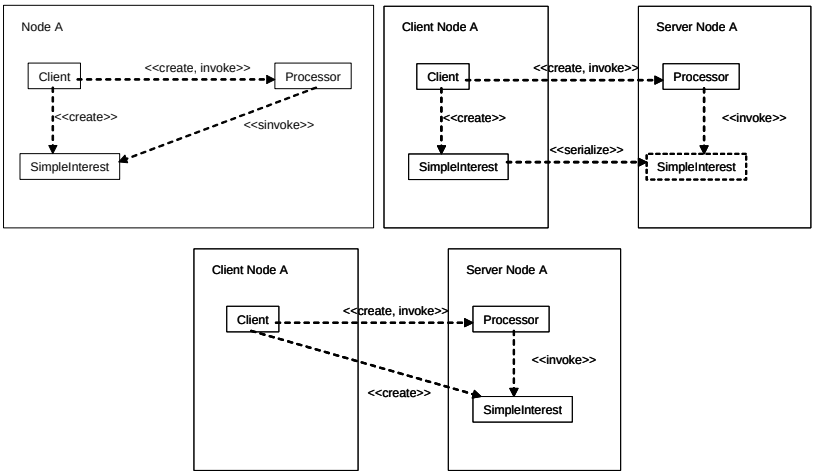


Figure 5 - 2 Node partitioning a) every objects reside in the same machine; b) Client and SimpleInterest reside in the same machine; c) Processor and SimpleInterest reside in the same machine

The second problem is in the middleware choice. Suppose a bank used Java RMI as the only remote object distribution mechanism but later moved to J2EE EJB to use naming and transaction services offered by a J2EE container. However, once the bank offers the service to other banks or third party credit agencies, it might have to support CORBA or web service middleware in order to interoperate with them. The same servant may needs to have a Java remote stub, a CORBA object stub and a web service endpoint interface. As the number of servant objects grow, the possible permutation of the configurations is far too great to manage manually.

The third problem is with the object distribution strategy. Not all middleware support objects' behavior distribution due to an interoperability issue across different platforms. Target platform architecture can be different and may not be able to execute transported object code (b). Even though it is possible to work around by creating customized XML marshalling (in case of web service) or using CORBA externalization service to send class definition so that the transported objects can be reconstructed and executable at the target machine but this will limit the interoperability only to compatible platforms.

The fourth problem is in the distribution depth of objects' graph. Objects often associate with others; some are deeply related. Distribute the entire object graph does not yield

good performance while distribute only object references may increase network overhead if access to remote objects at the servant are frequent. Various work proposed strategies to distribute different levels of object graph from a fixed depth object graph distribution (shallow copy and n-depth copy) to a variable-depth distribution.

This chapter proposes a modeling technique that addresses these four problems. It combines the UML class diagram to represent class models, the new UML 2.0 Composite Structure diagram to define object relationship in logical partitions and our Distribution profile to indicate distribution strategy. We will show that it is possible to abstract the distribution concern out of the object models and how the distribution-independent object models are translated into implementations. Next section shows tedious and error prone steps of manual translation of the example class model in using different middleware.

5.2 Middleware Specific Implementations of the Case Study

This section shows how the simple class model in the last section is translated into implementations using four middleware services; Java Remote Method Invocation (RMI); Java Enterprise (J2EE), CORBA and web service. This task is often done manually or semi-manually with the assistance from development tools. However, once the class model is translated to use a middleware service, the number of implementation artifacts grows and it is difficult to migrate from one implementation to another.

When the developer started implementing the class model in to support distribution in scenario from b, he initially chose the Java RMI. The RMI provides ways to define remote interfaces and object passing across different Java virtual machines. It requires specific design guidelines and code patterns to facilitate a creation of remote proxies, object marshalling and remote exceptions. Below are the refinement steps to the original class model during both development and runtime.

1. Partition classes into two packages; one contains client classes that invokes remote interfaces and the other contains server classes that implement remote interfaces.
    a. At the server package:

i. Change `Processor` from class to interface. Make the interface extends `java.rmi.Remote` and for every remote methods. Define the methods in the interface and make them throw `java.rmi.RemoteException` in their implementation.
ii. Create a `ProcessorImpl` class that extends `java.rmi.server.UnicastRemoteObject` and implements the `Processor` interface. Make the constructor throws `java.rmi.RemoteException`.
iii. The Task interface needs to extend `java.io.Serializable`.
iv. Add bootstrap code to the `ProcessorImpl` to create an RMI security manager, instantiate the `ProcessorImpl` object, bind it to a name and register the name to the RMI registry.
v. Create the `ProcessorFactory` class that extends `java.rmi.server.UnicastRemoteObject` and add `create()` and `destroy()` method. This class will be responsible for the creation and destruction of the `Processor` object.
vi. Run the RMI compiler against the `ProcessorImpl.class` and `ProcessorFactory.class`. It will generate a `ProcessorImpl_Stub.class` and `ProcessorFactory_Stub.class` which acts as remote proxies to the `ProcessorImpl` and `ProcessorFactory` respectively.

b. At the client package:
i. Create an RMI security manager.
ii. Obtain the RMI server URL from an input argument and lookup the `ProcessorFactory` stub from the server.
iii. Invokes a static method of `ProcessorFactory` to create a new instance of the `Processor` and return the `Processor` stub to the client.
iv. Pass the `SimpleInterest` and invoke the server stub `execute()` method.
v. Catch a remote exception
vi. Invoke the static method of the `ProcessorFactory` to destroy the server object.

2. At runtime:
a. At server:

i. Start the `rmiregistry` daemon. Its default port number is 1099.

ii. Start the `ProcessorFactory` server from the bootstrap code.

b. At client:

i. Specify the RMI server URL as a parameter

ii. Execute the `Client` class

Figure 5 - 3 below is the class model of the RMI implementation.

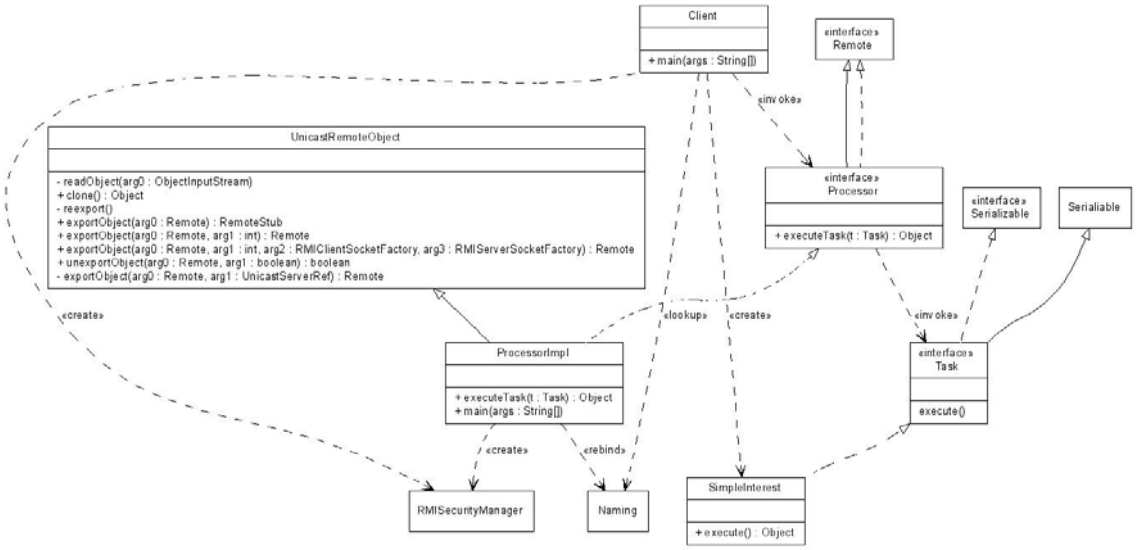


Figure 5 - 3 Class model using RMI distribution method

The Table 5 - 1 below compares the number of changes required from the first class model to the RMI implementation. For *n* remote class, there will be *n* interfaces, *2n* implementation and stub classes and optional *2n* factory classes if the client is responsible for the creation of the server object.

Item	*Simple*	*RMI Implementation*
Interfaces/Classes	1/3	2/3
Interfaces/Classes that require modification	-	3
Depended RMI library classes	-	4
RMI generated class	0	1 (`ProcessorImpl_Stub`)
Additional files		1 (`java.policy`)
Total lines excl. comments	36	84

Table 5 - 1 A simple and RMI implementation comparison

Note that the `ProcessorFactory` is not part of the standard RMI steps. RMI assumes that the server object is created by some process at the server and is ready when a client makes a remote invocation. However, in the example model, the client explicitly creates the server object and implicitly destroys the server object at the end of the client main method scope. The server factory object follows the *Factory* design pattern and is responsible for the server object life cycle. Since the server factory is a remote class, it also requires a stub.

If the same system is implemented in J2EE, the server object becomes an enterprise java bean (EJB). The naming service is performed by JNDI instead of RMI registry and the transport protocol choice is IIOP in addition to JRMP. From the sample model, the processor is a service object that does not have an object identity and therefore is modeled as a session bean. Each session bean requires a home class and a remote interface. The `Processor` becomes a remote interface and the `ProcessorBean` is a session bean that extends from `javax.ejb.SessionBean` and implements the interface. All remote interface must throw `java.rmi.RemoteException`. Passing objects need to be marked with `java.io.Serializable`. The deployment descriptor contains definition of all EJBs. The J2EE container will create a distributable stub for each bean. The client creates the server object by locating the session home from the JNDI and invoking the `create()` method to obtain the server stub. Each object parameter is passed by value if the object implements `java.io.Serializable` and passed by reference if the object implements `java.rmi.Remote`. By default, the entire object graph reachable by the parameter object will be marshaled. Custom object graph marshaling can be done by overriding the object's writeObject() and readObject() methods.

Both RMI and EJB supports object behavior passing across Java virtual machines. Therefore the `Task` concrete implementations can be instantiated, distributed and executed at the server location. However, this is not the case in environments that support only the object state passing but not the behavior passing such as web service. CORBA has an Externalization Service that supports custom object marshalling but it will only work if the target platform can execute remote object code which limits the interoperability goal.

The implementation of the externalization service varied, if not unavailable, in ORB vendors.

By default, RMI uses Java Remote Method Protocol (JRMP) as a binary transport protocol and thus limit the communication to be among Java programs. However, RMI can also be used to communicate with CORBA objects by using an IIOP protocol with some restrictions [GJSB05].

There are two ways to use RMI over IIOP. The first way is to use JNDI as a naming service. Instead of using `Naming.rebind(name, objRef)` to bind a server object reference to a name, we will first create a new JNDI initial context to bind it. The client obtains the server object reference from the JNDI name and invokes its remote method. Even though the IIOP is interoperable across CORBA implementations, the JNDI depends on Java and limits the applicability to other programming languages.

The second way is not to use JNDI. Then how would we transport the server Ties to the client? One way is to store the Tie files into a network file system accessible to both client and server, another is to convert them into an interoperable object reference (IOR) string using orb.object_to_string(objRef), or we could use a CORBA naming service (CosNaming). The following steps show how to implement the system using the network file system.

1. All remote implementations, e.g., `ProcessorImpl`, must inherit from `javax.rmi.PortableRemoteObject` instead of `java.rmi.server.UnicastRemoteObject`.
2. At the server, use the `javax.rmi.PortableRemoteObject.toStub()` to create a server stub from a server object reference (servant); connect the stub to a CORBA ORB and export the stub into an output stream.
3. At the client side, deserialize the server stub from the stream and obtain the server object reference from the ORB. Instead of using `Naming.lookup()` to resolve the server object reference, use `javax.rmi.PortableRemoteObject.narrow()`.

4. Regenerate the RMI proxies. CORBA requires a stub (a local proxy for a remote interface) and a tie (a server proxy that processes incoming calls and dispatch the calls to the server implementation class). The RMI compiler generates the stub and tie classes from the `-iiop` option. In the example, RMI will generate `_ProcessorImpl_Tie.class` and `_Processor_Stub.class`. If the client is implemented in other programming language, use the `-idl` option to generate a CORBA Interface Definition Lanague (IDL) file.
5. Start the CORBA ORB daemon `orbd` with a port number. Default port is 1050.
6. At the beginning of the client or server execution, define the initial naming factory to `com.sun.jndi.cosnaming.CNCtxFactory` and point to the ORB naming provider in step 6 using the IIOP URL, for example, `iiop://127.0.0.1:1050`.

To use IOR, the servant inherits from `org.omg.CORBA.Object` and the remote object lookup is performed by `orb.string_to_object(iorString)`.

A class model of a CORBA implementation is shown below in Figure 5 - 4.

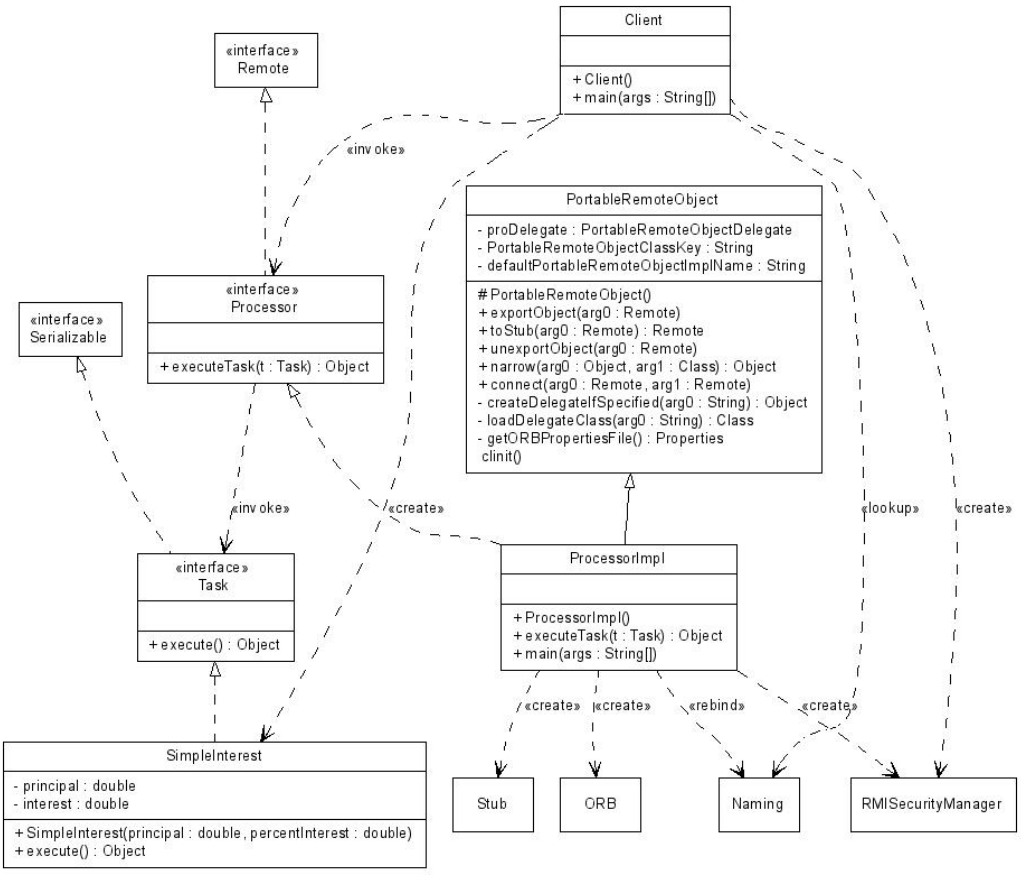


Figure 5 - 4 CORBA implementation

For CORBA, essentially the following steps have to be performed:

1. Initialize the ORB.
2. Obtain a reference to the root naming context by requesting the ORB to resolve_intial_references.
3. Narrow the reference down from a CORBA object reference to the actual NamingContext class.
4. Obtain a reference to the required CORBA object by querying the naming context for the object name.
5. Narrow the reference down from a CORBA object reference to the actual class.
6. Invoke the method on the object reference as if it were the object itself.
7. For every subsequent method call, repeat step 6.

Note that in the CORBA case, every step is a single statement.

```
URL url = new URL("http://localhost:8080/apache-soap/servlet/rpcrouter");
Call call = new Call();
call.setTargetObjectURI("urn:Hello");
call.setMethodName("sayHelloTo");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
Vector params = new Vector();
params.addElement(new Parameter("name", String.class, "Mark", null));
call.setParams(params);
Response resp = null;
try {
  resp = call.invoke(url, "");
  if ( !resp.generatedFault() ) {
    Parameter ret = resp.getReturnValue();
    Object value = ret.getValue();
    System.out.println(value);
  }
  else {
    Fault fault = resp.getFault();
    System.err.println("Generated fault!");
  }
}
catch (Exception e) {
  e.printStackTrace();
}
```

Listing 5 - 1 CORBA standard implementation

Compare this to a CORBA invocation in Java:

```
try {
  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
  org.omg.CORBA.Object rootObj = orb.
resolve_initial_references("NameService");
  NamingContextExt root = NamingContextExtHelper.narrow(rootObj);
  org.omg.CORBA.Object object =
root.resolve(root.to_name("AcmeMyService"));
  MyService myService = MyServiceHelper.narrow(object);
  int ret = myService.sayHelloTo("Mark");
} catch (MyServiceException e) {
  System.err.println("Generated fault!");
} catch (Exception e) {
  e.printStackTrace();
}
```

Listing 5 - 2 CORBA RMI/IIOP implementation

5.3 Object Distribution Profile

The object distribution profile defines 4 types of objects; a value object, a depended object, an entity object and a service object. A *value object* is an object with no identity. It can be of primitive types or a structure of primitive values. The value object never changes its internal state and therefore is safe to copy and distribute. A *depended object* is an object that does not live alone. It is a part of another object. An *entity* is an object with identity and lives beyond the execution environment. It is often stored in a persistent storage and may contain value or depended objects. A *service object* is an object that provides service and does not move. The service object can have an identity if each instance of the service must be uniquely identified and looked up or it can have no identity if the service object is singleton, replaceable or shared in a pool. For example, an `Order` object is an entity that can be identified by a key such as an order `Id`. The `Id` can be natural or surrogate and is usually a value object. The `Order` object consists of a set of `LineItem`s that are depended objects. `LineItem`s do not have identities and must belong to an `Order`.

Classification of objects facilitates the object passing semantics. There are 3 notions of object passing; pass-by-reference, pass-by-state-value, a pass-by-object-value. *Pass-by-reference* passes objects by location in memory if the caller and the callee reside in the same address space or by a globally unique id if they reside in different address space. Pass-by-state-value and pass-by-object value are variations of pass-by-value semantics where *pass-by-state-value* sends only the object and its depended objects' states while *pass-by-object-value* also sends the object and its depended objects' class definitions. Pass-by-object-value allows objects in the entire object graph to be reconstructed at the receiver end so that they can be executed locally. It is a semantics needed to support mobile code. The separation of pass-by-state-value and pass-by-object-value allows for the verification whether the object value passing is permitted in the target platform environment. For example, we cannot pass Java objects to be executed at a CORBA C++ server.

Value objects are always passed by state value. Passing entities and depended objects are tricky. Depended objects cannot be passed directly, that is, as a root of the object graph. Entities and depended objects can be passed by reference or value. If they are passed by reference, only their stubs are passed. If they are passed by state-value, only their states in their object graph are sent. If they are passed by object-value, their states and class definitions of the reachable objects in the graph are sent. The object-value passing will work only if the target environment is the same as the caller's.

The service object may be called locally or remotely or both. A decision whether the object invocation is local or remote depends on the execution boundary between the caller and the callee. From the modeling point of view, developers should not be concerned with identifying which invocations are local or remote in the object models. Instead, the model transformer tool should infer the type of invocations when the developers define executing boundary in composite structure models.

To invoke a service object, it is necessary to locate the object. The object can be located by a name and the binding between the object and the name is managed by a naming service. A fully qualified name of the service consists of the location of the service ({node}), the name of the service ({name}) and the optional instance identifier ({id}) of the service if the service has an identity. If the service is shared or a singleton, the instance identifier is ignored. The service object is tagged with «service». If the service object is created on-demand by a client, it will be destroyed at the end of the client block. If the service object is a singleton and lives as long as it environment is running, it will have an «active» tag and the bootstrap code is added to instantiated the service object when the environment starts. If the service object can be pooled, its life cycle willed by managed by the environment pool manager.

Internally, a class method has a stereotype «local» and «remote» to indicate whether the method can be invoked locally (default) or remotely or both. Instead of declaring every single method in a class as «local» or «remote», it is desirable to group remote methods

into an interface and tag the interface with «remote». All methods defined in the interface will inherit the «remote» automatically.

Viewpoint defines the scope and visibility of object modeling. A distribution-independent viewpoint shows the class diagram that does not show the distribution concern (Figure 5 - 5). A middleware-independent distribution viewpoint shows the same model with distribution concern but do not show specific artifacts from a specific middleware. An implementation viewpoint is the executable model that uses a middleware. Figure 5 - 5 and Figure 5 - 6 below are examples of different viewpoints from distribution-independent to middleware-independent views.

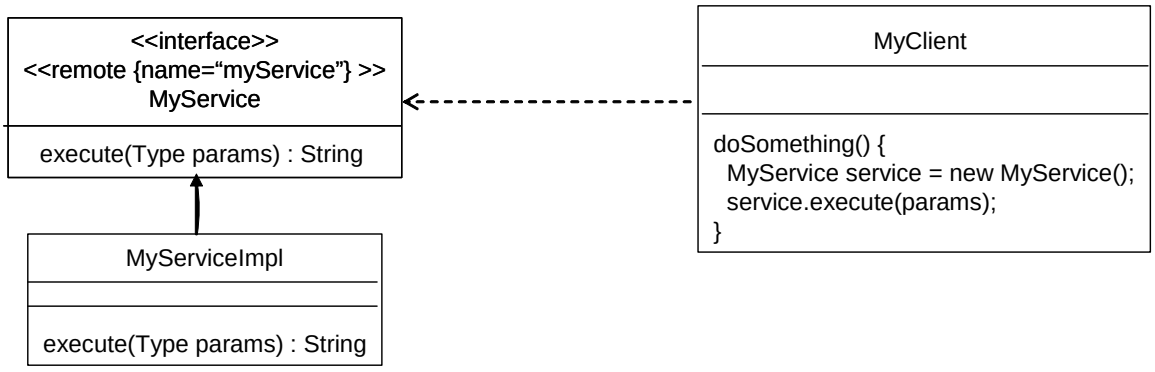


Figure 5 - 5 PIM with a remote name

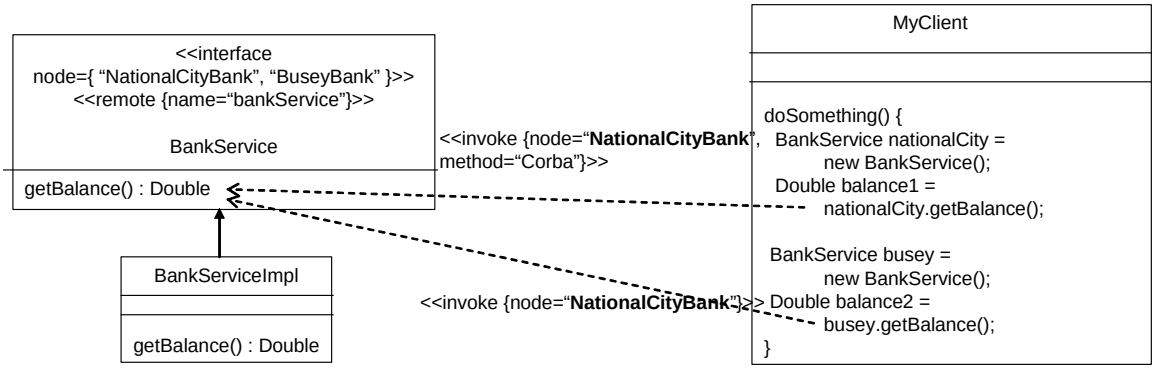


Figure 5 - 6 PIM with node names

Table 5 - 2 below is the summary of the distribution profile.

Stereotype	*Applies to*	*Attributes*	*Description*
«import»	Model	{profile=”distribution”}	Allows models to use the distribution service.
«remote»	Classifier, Operation	{name : String[0..*]}	If it is applied to an operation, it indicates that the operation is remote. If it is applied to a classifier, all class operations are remote. This stereotype is mutually exclusive to «local».
«local»	Classifier, Operation		If it is applied to an operation, it indicates that the operation is local. If it is applied to a classifier, all class operations are local. This stereotype is mutually exclusive to «remote».
«service»	Classifier		A service class.
«node»	Package, Classifier	{name : String[0..*]}	A logical named node. If it is applied to a classifier, its instance will be created from this node. If it is applied to a package,

			instances from all containing classes will be created from this node.
«invoke»	InvokeAction	{node : String, method : String}	Invoke an instance operation of the specified node. If the method field is empty, it will use a method defined globally.

Table 5 - 2 Distribution Profile

5.4 Summary

The distribution profile provides middleware transparency. Object models need not include distribution concern. Distribution is important because it increases the understandability of the object model because it abstracts out the middleware specific details and it reduces migration cost from systems that need to be migrated to or integrated with middleware-incompatible systems.

Chapter 6 Transaction Service

6.1 Introduction

Transactions are essential for building reliable, large scale enterprise applications. They usually involve persistent objects that need to be updated in an atomic way. This chapter builds on the persistence service and defines a unit of work that deals with objects that participate in transactions. The transaction service extends persistence objects defined in chapter 3. It calls a transactional persistence object an *entity*. The service tracks entities involved in a unit of work operation and makes sure that the persistent state of these entities depends on the completion status of the transaction.

Transactions are an example of how one profile can depend on another. This makes the definition of the profile simpler but the definition of its transformations more complex. The transformation must deal with persistence entities participating in each transaction, transaction scope and concurrency control. This is not trivial because the data sources of participating entities are not known at the PIM level. Moreover, different middleware enforce different ways of persisting objects, managing transaction scope and specifying concurrency levels. Using the unit of work modeling, developers define transactional operations in PIMs and specify entity data source locations in a separate annotation file. Transaction service transformers use this information to track entities participating within the transaction operations and obtain entity data sources from the annotation file to generate a transaction middleware specific implementation. These chapter shows two different transformations of the same PIM; one using a local transaction manager from a Hibernate session and the other using a distributed transaction manager from a J2EE JTA [Sun06].

6.2 A Motivating Example

Consider an e-commerce order example adapted from iBATIS's JPetStore [Jpe02]. A simple order system contains an *insertOrder* method that updates two kinds of objects as

a single unit of work. The method creates an order containing all line items and updates the quantity of each line item from the item inventory. These updates must be atomic to ensure that both updates succeed or not at all. A simple specification will look like Listing 6 - 1 below.

```
/**
 * @transaction
 */
public void insertOrder(Order order) {
   order.setOrderId(getNextId("ordernum"));
   orderDao.insertOrder(order);
   itemDao.updateQuantity(order);
}
```

Listing 6 - 1 An insertOrder method example

JPetStore uses the Data Access Object (DAO) pattern [AMC03] as a resource manager to manipulate object data. The method acts as a unit of work such that all changes to the order and line items must be stored in persistence storage or the transaction must be rolled back to the original state. The specification above uses a JavaDoc @transaction comment to indicate that this method is transactional but does not show how the transaction is implemented, where exceptions are handled or where the data are stored. The actual implementation must address these issues.

If the order and inventory data reside in the same database, they can be accessed atomically by creating a transaction from the database transaction manager; data manipulation is processed in the same transaction context. This transaction is considered local because it is managed by a single transaction manager. However, if this e-commerce system also sells suppliers' products as well as its own or the system moves the inventory data into another database then it will use more than one database and so the *insertOrder* must support distributed transaction. Distributed transaction guarantees the consistency of database operations across multiple database boundaries but requires high overhead in distributed synchronization mechanism and should be used only when necessary. Listing 6 - 2 shows an implementation of a local transaction provided by

iBATIS' DAO framework while Listing 6 - 3 uses the Java Transaction Service (JTS) to implement a distributed transaction.

```
private void insertOrder(Order order) throws DaoException {
  try {
   storeDaoManager.startTransaction();
   order.setOrderId(getNextId("ordernum"));
   orderDao.insertOrder(order);
   itemDao.updateQuantity(order);
   storeDaoManager.commitTransaction();
  } catch (Exception e) {
    // omit roll back exception handling
    storeDaoManager.rollbackTransaction();
  }
}
```

Listing 6 - 2 The insertOrder with local transaction

```
private void insertOrder(Order order) throws DaoException {
  try {
   javax.transaction.UserTransaction ut = context.getUserTransaction();
   ut.begin();
   order.setOrderId(getNextId("ordernum"));
   orderDao.insertOrder(order);
   itemDao.updateQuantity(order);
   ut.commit();
  } catch (Exception e) {
    // omit rollback exception handling
    ut.rollback();
  }
}
```

Listing 6 - 3 The insertOrder with distributed transaction

Notice that both the DAO and UserTransaction require a declarative transaction policy that is not shown in the listing. The distributed implementation requires a J2EE container to use the JTS.

There are two important points from the examples above. First, the developer has to support both local and distributed transactions even though the distribution decision may not be made until the software is deployed. Second, both implementations depend on

how the middleware supports transactions and thus makes it harder to change the code from one implementation to another, i.e., moving from JTS to CORBA Transaction Service. We want to make the *insertOrder* independent of data sources and transaction support policy so that the platform independent implementation is as simple as that in Listing 6 - 1. We will show that a mapping tool can generate different PSMs from the same PIM with each specific annotation. However, before doing so, it is important to describe a transaction model and how to support transactions in a platform independent way.

6.3 Transaction Service Characteristics

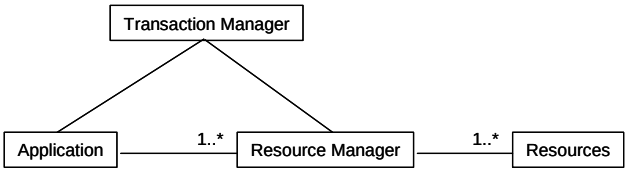


Figure 6 - 1 Transactional Model

Figure 6 - 1 shows an application transactional model. An application accesses data from one or more resource managers. A resource manager manages a set of data that can be read or written by providing data source interfaces to resources. Each data source provides a connection to resources and the connection can be pooled. Typical resource managers are a RDBMS, an ODBMS, or an XML DBMS. If the application needs to access multiple data sources atomically, it should obtain a transaction manager from the resource manager to manage the transaction boundary of the atomic operations. There are four primary concerns in supporting high level, platform independent transactions.

1. *Local/global transaction scope. A transaction scope* indicates the type of the database management system [BGS92]. The scope of a transaction is local if the transaction manipulates objects that are managed by a single resource manager. If the resource manager is a DBMS, the transaction is handled by the transaction support from the database connection. On the other hand, if the transaction manipulates objects from different resource managers, the scope of a transaction is global. A distributed transaction manager is needed to coordinate changes to objects from

different resource managers. These resource managers must support distributed transaction commit and rollback such as two-phase commit [BHG87] to ensure data consistency across objects in the transaction. For example, a fund transfer system withdraws money from an account from one bank and deposits the same amount minus a processing fee into another account in another bank (the processing fee initiates a nested transaction that deposits it into the processing bank's own account). Since each bank has its own database, it is not possible to use one database manager to guarantee atomicity and consistency across multiple databases.

Transaction scope support at the PIM level implies that the model mapping tool must derive the data source of each object involved in a transaction. The data source of each object is defined at the class level in a persistence annotation. The tool finds objects accessible from the transactional operation in the PIM and obtains the data sources of the objects from the persistence annotation of the object's class to choose an appropriate kind of transaction. If objects reside in the same database, it's more efficient to use local transactions provided by the database manager. Otherwise, it is necessary to use a higher overhead, distributed transaction manager.

2. *Concurrency control.* Enterprise applications are used by many users simultaneously. One user should not see changes made by another until the other's changes are committed. For example, many customers order products from the e-commerce website and often they compete to purchase the same products. If they put goods in their shopping cart but have not yet placed an order, others will still see these goods available. Whoever places the order first will get the goods while others see the goods as back ordered. Therefore, developers need to isolate changes within one transaction from another. Concurrency control [BG81] is a mechanism to ensure serializability of concurrent transactions. The level of isolation determines the degree of concurrency. There are two general classes of concurrency controls; pessimistic and optimistic [MN82]. Pessimistic concurrency control uses locks to prevent concurrent accesses to shared resources. The type of locks (read/write) depends on the isolation levels specified by the developers. ANSI/SQL defines four standard levels; read

uncommitted, read committed, repeatable read and serializable. The impact of the isolation on different consistency phenomenal is shown in Table 6 - 1.

Optimistic concurrency control allows simultaneous accesses to shared resources and uses object snapshot or versioning to detect potential concurrency conflicts during the validation step (Figure 6 - 2). Supporting versioning usually requires modifying the object schema by adding a version field. The field can be either an incremental version number or a read/write timestamp [BHG87]).

Isolation Level	Consistency Phenomenal		
	Dirty Read	Non-repeatable Read	Phantom
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

Table 6 - 1 Isolation level impact on transaction consistency

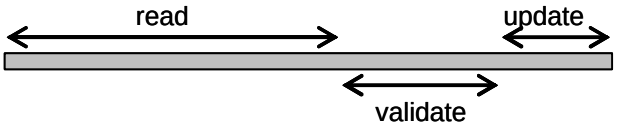


Figure 6 - 2 Three steps in optimistic concurrency control

Transaction support at the PIM level requires a model notation that indicates the type of concurrency control for each transactional operation. If pessimistic control is specified, an isolation level must be indicated. A model transformation uses this information to generate an appropriate isolation level for each transactional operation

in the target platform, for example, as a declarative transactional policy in the deployment descriptor file in J2EE platform or as an implementation configuration file in CORBA platform.

If the concurrency control is optimistic, the transformation tool must transparently create a version field for each participating objects in the PSM and use it to compare the object access conflict during the transaction validation step. A version field is added to the marked PIM but is not seen in the domain PIM.

3. *Transaction granularity.* A transaction is usually short and should be done within a single client-server invocation. However, there are cases where a transaction spans multiple requests or depends on user interaction. For example, an online web-based travel reservation wizard contains air, car and hotel reservation pages; a customer fills out information for each page and confirms the reservation at the last step. If the customer confirms, all reservations are stored; otherwise, they will be discarded and available to other customers. Transaction granularity indicates a life cycle of the transaction and can be per-request (Figure 6 - 3a), per-request-with-detached-objects (Figure 6 - 3b) or per-application-transaction (Figure 6 - 3c) [BG04]. The last two types are called long lived transactions [GS87].

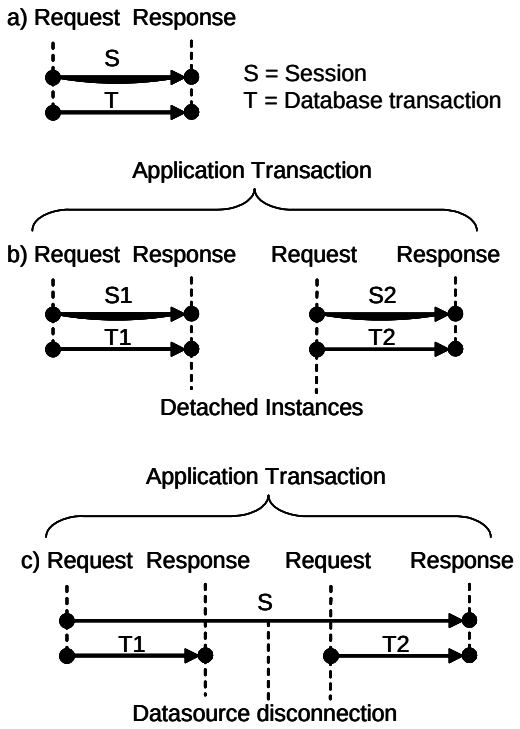


Figure 6 - 3 Transaction Granularity; a) per-request

b) per-request-with-detached-objects

c) per-application-transaction.

Transaction granularity support at the PIM level depends on the middleware used at the PSM level. In a per-request granularity, the transaction is obtained from a session context or from a database connection and the transaction terminates at the end of the request scope. However, if the transaction spans multiple requests and each request executes in separate sessions or in a stateless session environment, the target PSM must support disconnected persistence objects. Disconnection from persistence store allows objects to be accessed offline and synchronized back during reconnection. During disconnection, the system does not have to maintain the database connection.

On the other hand, if each request executes in a stateful session environment, the target PSM must support database disconnection and reconnection and the reference to the database connection must be stored in the stateful session.

If the transaction middleware does not support the transaction granularity specified at the PIM, the model transformation generates an error. The developer must choose an appropriate middleware or change the design that uses less restrictive granularity.

4. *Nested transaction*. Transactions can be nested inside a parent transaction at an arbitrary depth [Mos81]. If the parent transaction fails, all nested transactions including the parent will be rolled back. If a nested transaction fails, the parent transaction may decide to roll back only the nested transaction and proceed or roll back all transactions in the transaction tree.

   A PIM uses a nested transaction when a transactional operation invokes another transactional operation. Each operation contains a transaction attribute indicating whether the transaction will 1) obtain a new transaction from the transaction manager; if there is already a transaction, it will fail; 2) participate with the current transaction or create a new transaction or 3) create a nested transaction. The nested transaction depends on the support of the target PSM. If the target platform does not support nested transactions, the model transformation generates an error.

To address these four concerns, developers need a systematic way to define transaction support at a PIM level that does not depend on choices of transaction and persistence middleware. We propose Unit of Work (UoW) modeling as a solution to support transaction of persistence objects at a platform independent level. The next section describes the UoW modeling concept as a way to specify transaction boundary, determine objects involved in the transaction and explains a transformation tool that takes an input PIM and annotations for target platform choices to create an executable PSM.

6.4 Unit of work Profile

Unit of work is a broad term that has many meanings. Fowler defines *a unit of work* (UoW) as a mechanism to maintain a list of objects affected by a business transaction and coordinate the writing out of changes and the resolution of concurrency problems [Fow03]. The UoW supports object transactions by determining transaction scopes from

one or more transactional operations; registering objects participating in the transaction; and commit/merge/rollback changes at the end of the transaction to a permanent storage (Figure 6 - 4). The granularity of data in UoW is at the object level. Therefore, the persistence store is not limited to a relational database. The UoW also supports in-memory, disconnected objects. A program can read objects from a database to main memory; detach the in-memory objects from the database; make changes to the in-memory objects; and finally reconnect and synchronize them back to the database. At the end of the transaction scope, the UoW commits the transaction by verifying and/or providing synchronization of objects in memory and those in persistence storage. It will fail if the objects are not consistent according to the concurrency policy and the transaction will rollback to the original state.

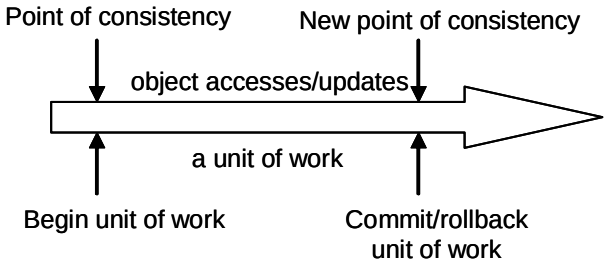


Figure 6 - 4 A Unit of Work

A program must use a database transaction manager to support in-memory disconnected objects. UoW registers objects read from the database and keep track of the original states of the objects (Figure 6 - 5) [Top06]. This can be done by using snapshots, timestamps or version numbers. In-memory objects can become disconnected from the database. Changes to the in-memory objects will not be seen from another transaction, thus providing transaction isolation. When the transaction commits at the end of the UoW, the registered objects are read from the database again and compared with the original states. If they are different, it means that an earlier transaction has changed the objects in the database and the later transaction will fail. UoW supports transactions that span multiple web requests by storing the UoW transaction manager in a session or an application context.

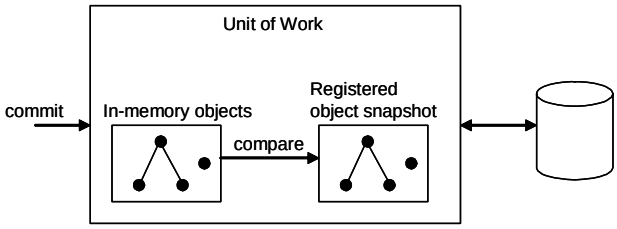


Figure 6 - 5 Unit of work keeps track of in-memory objects and original objects read from database

The UnitOfWork is a profile that supports transactions. The profile defines two stereotypes. The first is the «UnitOfWork» stereotype that applies the UML::Operation. A modeler uses «UnitOfWork» by marking the stereotype to the transactional operations of the PIM. The second is the «entity» stereotype. The «entity» is a specialized stereotype of the «persistence» from Chapter 3. It applies to a class who instances are persistent and must participate in transactions. Updates to entities within a Unit of Work operation occurs at the end of the transactional operation if the transaction commits or does not occur at all if the transaction aborts. Since the UML uses a dependency relationship notation to specify relationship between model elements, it is possible to automatically derive persistent objects involved in any transactional operation by traversing the dependency graph from the transactional operations to persistent objects reachable from them.

Modelers annotate the transaction properties of the operation by specifying «UnitOfWork» stereotype attributes (Figure 6 - 6). Default attribute values are underlined. The *transactionAttribute* indicates whether the operation must require a new transaction or join an existing transaction. If there is already a prior transaction, a new transaction will be nested. The *concurrencyControl* indicates whether to use optimistic or pessimistic concurrency control. If it is optimistic, a mapping tool will generate versioning schema for persistent objects reachable from the operation. If it is pessimistic, the *isolationLevel* identifies the degree of concurrency. It can be read uncommitted, read committed, repeatable read or serializable.

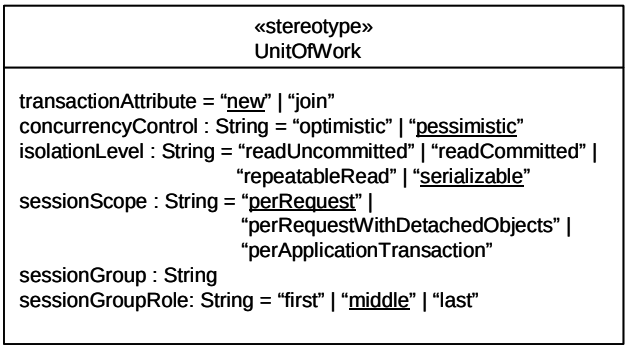


Figure 6 - 6 «UnitOfWork» stereotype attributes

The *sessionScope* indicates the boundary of the transaction, if the scope is per request, the transaction lives only in a local session thread. If the scope is per request with detached objects, the persistent objects can span multiple requests by detaching themselves from a database connection and reattaching back in a subsequent session. This option is possible only in a transaction middleware that supports detached objects. The last scope is per application transaction that uses an application-level context to maintain object lifecycle.

The *sessionGroup* is required if the operation is part of the unit of work sequence. The session group is a unique name for each unit of work and the *sessionGroupRole* indicates whether the operation is the first, middle or last step in the sequence. A model transformation tool uses the information from the session group role to generate code that begins or ends a transaction.

After a PIM is marked with the unit of work stereotype and annotated with transaction attributes, a mapping tool transforms it to a PSM as described in the next section.

Stereotype	*Applies To*	*Description*
«import {profile="UoW"}»	Model	Unit of Work service import
«entity»	Class	A class whose instances are persistent and participate in transactions.
«UnitOfWork»	Operation	A transactional operation that all updates to entities must be done in an atomic way.

6.5 Mercator transformation for Unit of Work

A UoW transformer is triggered when the Mercator tree iterator visits each «UnitOfWork» operation. The transformer analyzes the dependency relationship of the operation and collects objects involved in the unit of work. The transformer determines objects participating in the UoW by checking the UML dependency relationship graph from the UoW operation and obtaining persistent objects reachable from the graph. Persistent objects may be reachable directly from the operation or can be reachable via DAO as is the case in Listing 6 - 2 and Listing 6 - 3.

Note that persistent objects that participate in transactions are marked with «UoW::entity» which is a specialized stereotype from the «Persistence::persistence». For each entity object, the transformer will use the annotation to check the data source. If all objects involved in a unit of work have the same data source, the unit of work operation uses the data source's transaction manager. If persistent objects are defined in a container managed environment, the unit of work uses the container managed transaction. If they are from different data sources or managed containers, the unit of work uses a distributed transaction manager that supports two-phase commit. In Java, a distributed transaction manager is implemented by Java Transaction API (JTA) with data sources that support extended architecture (XA). If the target model required a distributed transaction manager but the data source does not support XA, the PIM-to-PSM transformation raises a transformation error.

The isolation level attribute in the «UnitOfWork» is used to define transaction property for the unit of work operation. The UnitOfWorkTransformer assigns the transaction scope in a deployment descriptor file (as in the JTAUnitOfWorkTransformer) or a configuration file (as in the HibernateUnitOfWorkTransformer). It is possible to create other transformer subclasses to define the transaction scope as comment attributes (Listing 6 - 1) used in XDoclet or as Java annotations in J2SE 5.0. The UnitOfWorkTransformer will create warnings if there are «Persistence::persistence» objects in the «UnitOfWork» operation. Since «Persistence» objects do not participate in

transactions, this may be desirable but the warnings will at least raise issues whether the «Persistence::persistence» objects must participate in the transaction and should be promoted to «UoW::entity» objects.

6.6 Case study

We will use the e-commerce order system in the motivating section as our case study. We developed two transformers for UoW; one for Hibernate UnitOfWork transformer and the other for J2EE UnitOfWork transformer. Figure 6 - 7 shows a partial PIM of the JPetStore sample application. The PIM imports the Persistence and the UnitOfWork profiles. The *insertOrder* method updates the Order and Item objects via OrderDao and ItemDao respectively. The updates must be atomic so the modeler marks the method with the «UnitOfWork» stereotype. Item and Order are transactional persistence objects and are marked with «Entity».

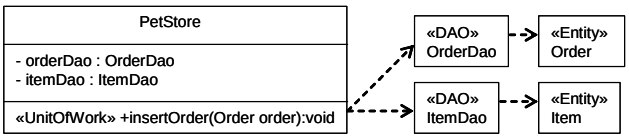


Figure 6 - 7 The JPetStore PIM

We will address the four issues in section 6.3 as follows:

1. Local/Global transaction boundary. The modeler indicates the data sources of the Order and Item in the stereotype attribute of «entity». The petstore.persistence.annotation.xml in Listing 6 - 4 below defines two data sources.

```
petstore.persistence.annotation.xml
<Annotation id="dataSource1">
 <property name=”class”>org.apache.commons.dbcp.BasicDataSource</property>
 <property name="driverClassName">org.hsqldb.jdbcDriver</property>
 <property name="url">jdbc:hsqldb:hsql://localhost:9002</property>
 <property name="username">sa</property>
 <property name="password"></property>
</Annotation>
```

```
<Annotation id="dataSource2">
 <property name=″class″>org.apache.commons.dbcp.BasicDataSource</property>
 <property name="driverClassName">sun.jdbc.odbc.JdbcOdbcDriver</property>
 <property name="url">jdbc:odbc:myDSN<</property>
 …
</Annotation>
```

Listing 6 - 4 DataSource definition

Next, the modeler specifies the data sources for the Order and Item. If both Order and Item are stored in the same data source, the data source annotation of each entity will be the same as in Figure 6 - 8.

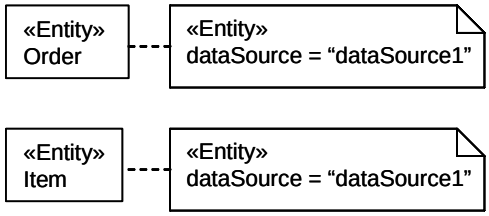


Figure 6 - 8 Order and Item are stored in the same data source.

On the other hand, if the Item data is stored in another data source, the data source annotation is modified as in Figure 6 - 9.

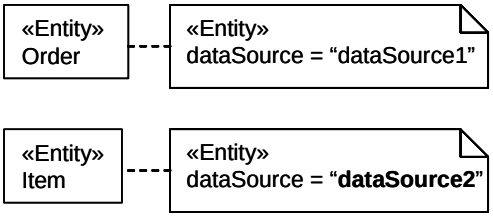


Figure 6 - 9 Order and Item are stored in different data sources

During the mapping, Mercator finds Order and Item from the dependency graph reachable from the *insertOrder* method via OrderDao and ItemDao respectively (Figure 6 - 7). It obtains their data source names from the annotation to determine whether the transaction should be local (Figure 6 - 8) or global (Figure 6 - 9).

2. Concurrency Control. A serializable isolation level ensures that the updates in the *insertOrder* method can be called simultaneously by different customers. However, the isolation level can be relaxed since there is no second read in the transaction; the

unrepeatable and phantom reads are not possible. Therefore the method isolation level is set to readCommited (See Figure 6 - 10).

3. Transaction Granularity. Since the transaction covers only one method invocation, its granularity is specified in the session scope as perRequest. The session step is omitted since the transaction does not span multiple methods.

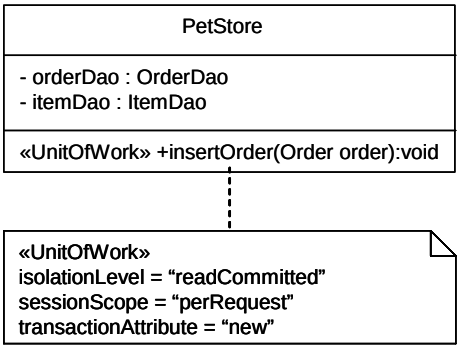


Figure 6 - 10 UnitOfWork attributes

4. Nested Transaction. This method is not part of another transaction and does not contain nested transaction. Therefore, the transaction attribute is set as new.

After the modeler finishes marking and annotating the PIM, he needs to specify a model-wide mapping choice for mapping process. The first implementation uses Hibernate persistence technology. He specifies the choice with annotations (Listing 6 - 5).

```
petstore.annotation.xml
…
<Annotation id="entity">
 <mapping choice="hibernate"/>
</Annotation>
<Annotation id="transaction">
 <mapping choice="hibernateTransaction"/>
</Annotation>
```

Listing 6 - 5 Add mapping choices for entity and transaction.

The HibernateUnitOfWorkTransformer takes the input PIM and annotation files to generate a PSM. Suppose the program uses the same data source for both Order and Item

(Figure 6 - 8), the HibernateUnitOfWorkTransformer uses a local UoW mapping. Listing 6 - 6 shows the generated *insertOrder* from Hibernate persistence and transaction. The original code from Listing 6 - 1 is shown in bold. The Hibernate transformer creates a local transaction from two generated *session* and *sessionFactory* fields.

```
public void insertOrder(Order order) throws DAOException {
  try {
   session = sessionFactory.openSession();
   session.beginTransaction();
   order.setOrderId(getNextId("ordernum"));
   orderDao.insertOrder(order);
   itemDao.updateQuantity(order);
   session.commitTransaction();
  } catch (Exception e) {
   session.rollback();
   throw new DAOException(e.toString());
  } finally {
   session.flush();
   session.close();
  }
}
```

Listing 6 - 6 The insertOrder in a local transaction using Hibernate

The same PIM can be generated into a different implementation. For example, to change from a hibernate transaction to J2EE JTA, the modeler copies the petstore.annotation.xml to petstore.annotation2.xml and changes the transaction mapping choice from hibernate to j2ee_jta (Listing 6 - 7). Suppose the data sources are distributed, the modeler annotates the model with different data sources as shown in Figure 6 - 9.

```
petstore.annotation2.xml:
…
<Annotation id="transaction">
 <mapping choice="j2ee_jta"/>
</Annotation>
```

Listing 6 - 7 Use J2EE to implement persistence and transaction

Since the data sources for Order and Item are now different, the JTAUnitOfWorkTransformer uses a distributed transaction mapping and creates a user

managed transaction from the generated *context* field. Listing 6 - 8 shows the generated *insertOrder*. Again, the original code from Listing 6 - 1 is shown in bold.

```
public void insertOrder(Order order) throw DAOException {
UserTransaction ut = context.getUserTransaction();
  try {
    ut.begin();
    order.setOrderId(getNextId("ordernum"));
    orderDao.insertOrder(order);
    itemDao.updateQuantity(order);
    ut.commit();
  } catch (Exception e) {
    ut.rollback();
    throw new DAOException(e.toString());
 }
}
```

Listing 6 - 8 The insertOrder in a distributed transaction using J2EE JTA

Both implementations throw a runtime DAOException. The exception subclasses from the HibernateException in Hibernate or the EJBException in J2EE. When a runtime exception is thrown, the transaction automatically rollbacks.

In addition to the two transformers, a transformer developer can create other mappings, for example, for JDBC, ORM or specialized implementations by developing new transformers, adding transformation choices to the UoW profile and reloading the profile to the model.

6.7 Summary

This chapter describes the unit of work modeling to support transparent transaction management in platform independent models and transforming the platform independent models that use unit of works into two implementation models. The UoW transformers determine the data sources used within transactional operations from class models and transaction properties from annotation files and generate appropriate transaction management implementations [WJ05]. The prototype currently supports basic

implementation for Hibernate and JTA but the Mercator transformation framework allows pluggable transformers to support other transaction middleware implementations.

Chapter 7 Messaging Service

7.1 Introduction

Application integration is a big issue in enterprise systems. Companies usually face problems in exchanging data between their suppliers and customers since each stakeholder has different ways of representing and communicating data. One of the key enablers for application integration is messaging. Messaging is middleware that handles information passing between disparate applications. Messaging middleware receives messages from a sender, stores them in reliable storage, optionally changes message format, and delivers the messages to one or more receivers. Messaging acts as a gateway between two different systems so that they can exchange information stored in different formats.

Unlike object distribution, messaging does not require a sender and a receiver to be running at the same time. If the receiver crashes, the messaging middleware will store the message and reliably deliver it when the receiver restarts. This feature provides *reliability*. Developers should be able to specify a reliability policy that specifies whether messaging middleware must deliver the message to the receiver exactly once, at most once, or at least once. The sender does not have to wait the message to be delivered to the receiver. Therefore, messaging provides *non-blocking call* service. The sender can choose whether to send messages without acknowledgement (fire-and-forget), or with responses via polling or via callbacks. The non-blocking call improves *performance*. Messaging uses channels to decouple senders and receivers. A channel is a logical path that senders connect and receivers subscribe to. A sender can send messages to a channel without knowing how many receivers are currently subscribing. The messaging middleware ensures that either all subscribed receivers got messages or none of them did. Since messaging middleware can process messages, it can reformat the messages suitable for each receiver and thus allows disparate applications that use the same data in different formats to exchange data without changes to the applications.

There are many message-oriented middleware in the market, for example, IBM WebSphere MQ, BEA Tuxedo, Tibco Rendezvous, Microsoft MSMQ, SonicMQ, FioranoMQ among others. Some use proprietary messaging protocols while others are built from the Java Message Service (JMS) or on top of JMS (J2EE Message Driven Bean) or using different mechanism (the webservice SOAP messages). Some companies forego these products and implement their own messaging library based on remote procedural calls. Even though there are many different messaging implementations, they serve the same information exchange goal. How do we allow different messaging middleware to communicate with one another, or better yet, is it possible to model applications that use messaging independent of message middleware?

This chapter introduces a messaging profile for middleware independent messaging service. At a PIM level, it defines stereotypes and APIs for applications to produce and consume messages. A PIM annotation indicates message format, delivery policy, and synchrony methods. A messaging transformer takes an input model that sends and receives object messages, an annotation that describes how messages are sent and a messaging middleware choice to generate executable PSM implementations. We will show three examples for a JMS, a Message-Driven Bean, and a web service implementation.

7.2 A Motivating Example

Suppose a simple inventory application sends a low inventory warning message to a supplier when a material is below a particular level. Ideally, it would be something as simple as:

```
supplierChannel = new Channel(“supplierForThisMaterial”);
if (inventory.isLow(aMaterial)) {
   Messaging.publish(supplierChannel, “Low:”+aMaterial.getString());
}
```

Listing 7- 1 Publishing a supplier message

This code does not know whether there are one or more suppliers. It only knows that if the inventory for a material is low, it will send a text message to every receiver currently subscribed to the *supplierForThisMaterial* channel.

To receive the message, the supplier should be able to write code as simple as:

```
public class Supplier {
    …
    supplierChannel = new Channel("supplierForThisMaterial");
    Messaging.subscribe(supplierChannel);
    …
    public void receiveOrderRequest(String requestInfo) {
           …
    }
}
```

Listing 7- 2 Subscribing the supplier message

The *receiveOrderRequest* method is a callback that is invoked when a message is sent through this channel.

In practice, messaging is much more complicated. We must specify which callback methods are for which channels. We must wrap message payload in middleware specific objects. For example, the Java Message System library (JMS 1.1) requires that the text message must be an instance of the javax.jms.TextMessage class and the channel must be either a *Queue* or a *Topic*. In this case, *Topic* is best since there is more than one supplier for this material. A message publisher must be created from a session. The session is created from a message connection which is obtained from a topic connection factory. Below is the JMS code for the sender:

```
if (inventory.isLow(aMaterial)) {
    Context ctx = new InitialContext(properties);
    ConnectionFactory factory =
           (ConnectionFactory)ctx.lookup("ConnectionFactory");
    Connection connection = factory.createConnection();
```

```
    Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
    TextMessage msg = session.createTextMessage();
    msg.setText("Low:"+aMaterial.getString());
    Destination destination = (Destination)
          jndiContext.lookup("supplierForThisMaterial");
    MessageProducer producer = session.createProducer(destination);
    producer.send(msg);
}
```

Listing 7- 3 JMS implementation for the message publisher

For a receiver, a straightforward JMS client can be written as:

```
public class Supplier {
    …
    Context ctx = new InitialContext(properties);
    ConnectionFactory factory =
          (ConnectionFactory)ctx.lookup("ConnectionFactory");
    Connection connection = factory.createConnection();
    Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
    Destination destination = (Destination)
          jndiContext.lookup("supplierForThisMaterial");
    MessageConsumer consumer = session.createConsumer(destination);
    consumer.receive();
}
```

Listing 7- 4 JMS implementation for the message consumer

Switching to another messaging middleware is non trivial. For example, to use the EJB Message Driven Bean, the message consumer must contain additional boilerplate code for EJB and implement a *MessageListener* interface.

```
public class SupplierBean implements MessageDrivenBean, MessageListener {
    protected MessageDrivenContext ctx;
    public void setMessageDrivenContext(MessageDrivenContext ctx) { … }
    public void ejbCreate() { … }
    public void ejbRemove() { … }
    public void onMessage(Message msg) throw JMSException {
```

```
            TextMessage textMessage = (TextMessage)msg;
            String text = textMessage.getText();
            …
    }
}
```

Listing 7- 5 EJB implementation for the message consumer

The text message for both JMS and MDB must be cast from a javax.jms.Message to javax.jms.TextMessage to String. The destination name and session properties must be specified in a deployment descriptor. The message consumer operation must be named *onMessage()* which is not meaningful in the context of the domain. Middleware independent messaging should allow developers to use meaning revealing names (e.g., receiveOrderRequest()).

Messaging in web service protocol is totally different. Developers must specify endpoint addresses with URL naming scheme. The message payload must be converted into XML data described by an XML schema. A primitive type payload can be mapped into XML's simpleType but more complicated object payloads are usually mapped into XML's complexType. The process of defining naming scheme and payload mappings are non-trivial and prone to error.

The above examples show that message contents need to be put into an envelope object (subclasses of the *Message* class) at the sender and retrieved back at the receiver. The extra code increases the complexity without incurring any benefit. Our aim is to abstract out the implementation details so that developers can write code as simple as that shown in the Listing 7- 1 which does not concern on middleware specific API. Developers should be able to take a platform choice and parameters in a separate annotation file and use messaging transformers to generate executable implementations. In order to design a middleware independent profile, we need to understand the properties and characteristics of a messaging service.

7.3 Messaging Characteristics

The example in the previous section shows many details for each implementation but they do not yet consider messaging styles, message payload, acknowledgement responses, message conversion, channel management and transaction support. These are important characteristics that need to be addressed in the platform independent messaging profile.

7.3.1 Messaging Styles

There are two primary styles of messaging; point-to-point and publish/subscribe. They differ in the number of message consumers. Both styles can have more than one message producer but if there is only one consumer, the style is point-to-point and the channel of communication is called a queue. If there are two or more message consumers, the style is publish/subscribe and the channel is called a topic. A consumer can peek into a queue but not into a topic. The messaging styles used to be important in JMS 1.0 but are unified in JMS 1.1. However, many existing systems are still using the old specification and some domains, i.e., embedded systems distinguish between them. A platform independent messaging profile must uniformly support both styles and provide a way to specify the channel type (topic or queue) for compatibility reason.

7.3.2 Message Payload

Platform specific middleware usually provides primitive data payload such as texts, bytes, maps or streams of primitive data. A more complicated payload is an object whose object graph is serialized at the sender and deserialized at the receiver. Messages in typical systems are usually non-trivial. A platform independent messaging profile must support payloads in both data formats, for example, EDI, ebXML as well as object formats. The message payload uses the persistence service provided by the persistence profile from Chapter 3.

7.3.3 Acknowledgement Responses

Message exchange between disparate systems is usually asynchronous compared to remote procedure calls in the object distribution service. The message producer sends

data to a messaging middleware and does not wait for a consumer to consume or return results. Volter defined 4 levels of message acknowledgements [VKZ04]. The Fire-and-Forget sends an invocation across the network and returns immediately. The Sync-with-Server sends an invocation and waits for an acknowledgement of the invocation from the receiver. The Poll Object creates a monitor at the sender that can be polled for the result of the invocation from the receiver. The Result Callback uses a callback object that is called from the receiver when the invocation is complete. The first two patterns do not expect results from the invocation while the last two patterns do. In fact the last two patterns are implementation specific and either one can be implemented from the other. We chose the Result Callback. The platform independent messaging must support Fire-and-Forget, Sync-with-Server and Result Callback.

7.3.4 Message Conversion

If a sender and a receiver use different message formats, the middleware must be able to convert from one to the other. Since messages are sent through the middleware channel, messages may internally be stored in a middleware internal message format. The in-filter and out-filter are interceptors that convert input messages from the sender and output messages to the receiver. If the middleware internal message format is identical to either an input or an output messages, the input or the output filter respectively may be null. The platform independent messaging must provide a way to specify message conversion filters as needed.

7.3.5 Channel Management

Developers identify channels by names. These names are logical and their binding to the channels (topic or queue) is managed by the Naming service that maps the logical names into real channel objects. The platform independent messaging service uses the Naming service from chapter 4 to allow logical name assignment for channels.

7.3.6 Transaction Support

Messaging operations can participate in a transaction. When a warehouse receives an order from a sales system, it prepares products and sends a message to inform a delivery truck. However, if the order is later cancelled, the message must not be sent. Therefore,

the message is part of the unit-of-work and must not be sent until the transaction is committed. If there are more than one message in the transaction, the middleware must guarantee that every messages are delivered or not at all. The platform independent messaging service must ensure the transaction support for messages.

7.4 Platform Independent Messaging Profile

There are 5 main participants in messaging; (1) a message producer asks (2) a messaging service to create (3) a named channel, assembles (4) an object payload and publishes it to the channel to that (5) one or more of message consumers subscribe. The Platform Independent Messaging Profile defines 6 stereotypes and 5 stereotype attributes. Four of the participants are defined as stereotypes and one as a stereotype attribute namely; (1) «producer», (2) «messaging», (3) {channel}, (4) «message», (5) «consumer». These stereotypes and the channel attribute belong to the Messaging namespace. A fully qualified name, for example, «Messaging::producer» will be used to uniquely identify the stereotype to prevent name clash.

A «producer» and «consumer» applies to classes that produce and consume messages respectively. A method that publishes a message is stereotyped with «publish» whereas a method that subscribes to a message is stereotyped with «subscribe». Both «publish» and «subscribe» methods must have a channel name.

```
«producer» class Inventory {
   …
   if (inventory.isLow(aMaterial))
          informSupplier(“Low:”+aMaterial.getString());

   «publish {channel=”supplierForThisMaterial”}» private abstract String
   informSupplier(«message»String aMaterial);
}
```

Listing 7- 6 Inventory publisher

The developer does not have to write code to set up channel connection, session, connection factory. This task is generated by a message transformer depending on the

message middleware choice. The message payload in the above example is a text string. However, it is not necessarily. The message can be of any object whose class is stereotyped with «message». The only constraint is that the payload type must be the same for the publisher and subscriber methods. Notice that the *informSupplier()* is an abstract method, since the implementation is generated by a messaging transformer.

The inventory object publishes product order messages to supplier objects. Therefore the developer indicates that supplier objects are consumers to the messages by marking the supplier class with «consumer» and the receiving method with «subscribe». This method contains only one message parameter that must be the same type as the sender's message.

```
   «consumer» class Supplier {
      «subscribe {channel="supplierForThisMaterial"}» public void
processRequest(«message»String text) {
      …
      }
   }
```

Listing 7- 7 Supplier subscriber

Notice that the message consumer method name does not have to be *onMessage()* as usually required by a messaging middleware. This flexibility allows for specifying useful names for the method. Again, the Supplier class needs not have code to set up channel, channel connection, session, channel factory or explicit channel subscription statement (Messaging.subscribe()). A messaging transformer is responsible for generating appropriate code depending on the choice of the messaging middleware. Since messaging method is usually asynchronous, the «subscribe» method return null. However, if the message producer expects a result, the transformer modifies the return statement into a message passing back to the producer's callback method of that channel.

```
   «producer» class Inventory {
      …
      if (inventory.isLow(aMaterial))
            informSupplier("Low:"+aMaterial.getString());
```

```
    «publish {channel="supplierForThisMaterial"}» private abstract String
                informSupplier(«message»String aMaterial);

    «callback {channel="supplierForThisMaterial"}» private void
          supplierResponse(«message»ResultCode code) {}
}
```

Listing 7- 8 Inventory publisher and callback

If the producer does not expect a result, it can choose Fire-and-Forget or Sync-with-Server by specifying the «publish» attribute with {ack={FireAndForget, SyncWithServer}}. By default, the acknowledgement style is SyncWithServer.

If the message consumer method returns a result, the return statement will be converted into a messaging response invocation back to the producer.

```
«consumer» class Supplier {
    «subscribe {channel="supplierForThisMaterial"}»
    public ResultCode processRequest(«message»String text) {
          …
          ResultCode code = …;
          return code;
    }
}
```

Listing 7- 9 Supplier returned result

The callback method is a dispatcher of messaging results for the channel. It is possible to have one callback method shared among multiple channels. It is the responsibility of the developer to design a message payload that contains a correlation id that the producer and consumer can use to identify the message during the callback. The messaging middleware invokes a callback method from another thread.

For systems that require explicit channel types, developers specify channel type attribute with {channelType = "topic" | "queue" }. By default, the channel type is topic. For example,

```
«publish {channel=”supplierForThisMaterial”, channelType=”queue”}»
private abstract String informSupplier(«message»String aMaterial);

«subscribe {channel=”supplierForThisMaterial”, channelType=”queue”}»
public ResultCode onMessage(«message»String text)
```

Listing 7- 10 Channel names and types

The transformer generates an error if the channel type of the *publish* and the *subscribe* methods do not match.

The message payload can be converted after it is sent by a producer and before it is received by a consumer. A «message» attribute { converter : MessageConverter } specifies a class name of the message converter that implements:

```
package org.mercator.profile.messaging;
public interface MessageConverter {
   public «message»Object convert(«message»Object);
   public «message»Object deconvert(«message»Object);
}
```

Listing 7- 11 Message converters

Messaging can be a part of a unit of work. For example, a unit of work consists of database operations and message transmission. We would like to make sure that all database operations and message delivery are successful when we commit the unit of work. This implies that the message sent by a producer must not be delivered immediately. Instead, the sent message is stored and waited at a messaging middleware. Once the unit of work commits, the message will be delivered. Messaging methods participate in a unit of work by importing the UnitOfWork profile into the model and put the «publish» and «subscribe» methods as parts of the «UnitOfWork» methods. Developers must be careful to use unit of work since it can lead to a potential deadlock. For example, a unit of work commits only when a message is successfully delivered but the message itself cannot be sent until the unit of work commits. A resolution is to use messaging middleware that supports two phase commit. At the prepare phase, the

message is delivered to a message consumer proxy. If the delivery is successful, the prepare phase acknowledges and is ready to commit the unit of work. If it is unsuccessful, the prepare phase sends a negative acknowledgement and the unit of work rolls back. During the commit phase, the proxy delivers the message to the message consumer.

The Table 7- 1 summarizes stereotypes and stereotype attributes in the Messaging profile.

7.5 Messaging Semantics

This section describes each stereotype in details and show how the stereotyped element is transformed into a JMS 1.1 implementation. Other transformations for EJB Message Driven Bean and Axis webservice are supported and briefly described in the next section.

Code examples in this section do not show exception handling. However, remote exceptions can occur. Chapter 2 describes a way to deal with application-defined and system exceptions. If developers care about particular exceptions, they need to define them as application exceptions and handle them explicitly; otherwise all system exceptions will be thrown as unchecked exceptions.

«producer»

A producer stereotype indicates that a classifier (a class or an interface) contains one or more operations that publish messages. A producer transformer must inject code to initialize a message connection and provide two private methods; a *_getContext() : Context* that returns a lookup context and a *_getMessageConnection() : Connection* that returns the current message connection. A publish method uses the *_getContext()* to obtains a reference to a named topic and the *_getMessageConnection()* to establish a session to create a message channel. This stereotype contains an attribute, *init,* which is either ‘pre’ or ‘lazy’. If the *init* is ‘pre’, the message connection is initialized during object creation. If it is ‘lazy’, the message connection is created when a first message is sent.

Name	*Applies To*	*Description*
«profileImport» {name=Messaging}	Model	Allows models to use the messaging service.
«producer» {init="pre" | "lazy"}	Classifier	Classifier that generates messages for one or more consumers. If the init attribute is "pre", the message connection will be created in a constructor. If it is "lazy", the connection is created when a first message is published.
«consumer»	Classifier	Classifier that subscribes to messages from a producer.
«publish» {channel=$name, channelType= "topic" | "queue", ack= "FireAndForget" | "SyncWithServer" }	Operation	Abstract method that publishes a message to a $name channel. The method must belong to a «producer» class. The message acknowledgement style can be fire-and-format or synchronize-with-server.
«subscribe» {channel=$name, channelType= "topic" | "queue", }	Operation	Method that receives the message sent from the message producer method of the $name channel. The method must belong to a «consumer» class.
«message» {converter=messageConverter*}	Classifier or Parameter	Method parameter that represents the message classifier. The message optionally contains a list of message converters. Each message converter implements the Messaging::MessageConverter interface.
«callback» {channel=$name}	Operation	Message producer's callback method used for obtaining a result from the message consumer of the $name channel.

Table 7- 1 Stereotype definition in the Messaging profile

Scenario 1: A producer class with the default *init='pre'*.

```
«producer {init='pre'}» class Inventory {
   private List products;
   …
   public Product findProduct(String upc) { … }
}
```

Listing 7- 12 Producer

will be transformed into code in listing 7-13. Notice that the original code in listing 7-12 is carried over and the generated code is added in italic.

```
public class Inventory {
   Properties _properties;          /* injected by container */
   Context _messageContext;         /* init by _initialize() */
   Connection _messageConnection;   /* init by _initialize() */
   private List products;
   …
   public Inventory() {
         _initialize();
   }
   private void _initialize() {
         _getMessageContext();
         _getMessageConnection().start();
   }
   private Connection _getMessageConnection() {
         if (_messageConnection == null) {
               Context context = _getMessageContext();
               ConnectionFactory factory =
               (ConnectionFactory)_ctx.lookup("ConnectionFactory");
               _messageConnection = factory.createConnection();
         }
         return _messageConnection;
   }
   private Context _getMessageContext() {
         if (_messageContext == null) {
               _messageContext = new InitialContext(_properties);
         }
         return _messageContext;
   }
   public Product findProduct(String upc) { … }
}
```

Listing 7- 13 Generated result of the Inventory producer

Scenario 2: A producer class with the *init='lazy'*.

```
«producer {init='lazy'}» class Inventory {
}
```

Listing 7 - 14 A producer with lazy initialization

The lazy initialization reuse the same transformation as in scenario 1 except that the *_initialize()* body is empty which prevents the program to pre-initialize the message connection and context.

```
public class Inventory {
   … // generated as scenario 1

   private void _initialize() {     // empty
   }
   … // generated as scenario 1
}
```

Listing 7- 14 Generated result with lazy attribute

Scenario 3: A producer interface.

```
«producer» interface InventoryInterface {
}

class Inventory implements InventoryInterface {
}
```

Listing 7- 15 Inventory class implements the Inventory interface

The producer transformer adds the *_getMessageConnection() : Connection* to the interface and injects code into all classes that implement this interface.

```
interface InventoryInterface {
   private Connection _getMessageConnection();
   private Context _getMessageContext() {
}

public class Inventory implements InventoryInterface {
   … /* generated as scenario 1 or 2 by the «producer» transformer */
}
```

Listing 7- 16 Generated result of the Inventory class

«consumer»

A consumer stereotype indicates that a classifier (a class or an interface) contains one or more operations that consume messages. Similar to the producer stereotype, a consumer transformer must inject code to initialize a message connection and two private methods; a *_getContext() : Context* that returns a lookup context and a *_getMessageConnection() : Connection* that returns the current message connection. A subscribe method uses the *_getContext()* to obtains a reference to a named topic and the *_getMessageConnection()* to establish a session to create a message channel.

Scenario 1: A consumer class.

```
«consumer {init='pre'}» class Supplier {
   …
}
```

Listing 7- 17 Supplier class with 'pre' initialized attribute

will be transformed into:

```
public class Supplier implements MessageListener {
   Properties _properties; /* injected by container */
   Context _messageContext;
   Connection _messageConnection;
   public Supplier() {
          _initialize();
   }
   private void _initialize() {
          _getContext();
          _getMessageConnection().start();
   }
   private Connection _getMessageConnection() {
          if (_messageConnection == null) {
                 Context context = _getMessageContext();
                 ConnectionFactory factory =
                 (ConnectionFactory)_ctx.lookup("ConnectionFactory");
                 _messageConnection = factory.createConnection();
          }
          return _messageConnection;
   }
```

```
    private Context _getMessageContext() {
          if (_messageContext == null) {
                _messageContext = new InitialContext(_properties);
          }
          return _messageContext;
    }
}
```

Listing 7- 18 Generated result of the Supplier class

Scenario 2: A consumer interface.

```
«consumer» interface SupplierInterface {
}

public class Supplier implements SupplierInterface {
}
```

Listing 7- 19 Supplier class without 'pre' initialized attribute

The consumer transformer adds the *_getMessageConnection() : Connection* to the interface, *_getMessageContext() : Context* and injects code into all classes that implement this interface.

```
interface SupplierInterface {
    private Connection _getMessageConnection();
    private Context _getMessageContext();
}

class Supplier implements SupplierInterface {
    … /* generated as scenario 1 or 2 by the «consumer» transformer */
}
```

Listing 7- 20 Generated result of the Supplier class

You will notice that the generated code from both the producer and the consumer transformers are very similar. The consumer transformer is in fact the same as the producer transformer with init="pre". In fact, both transformers inherit from the same abstract base transformer.

«publish»

A publish stereotype applies to an abstract operation that publishes a message. This operation must contain one parameter that is a message object. The message object can be a user defined object or a primitive data type. If it is a user defined object, the user defined class must have «message». If it is a primitive data type such as UML::String, we cannot put «message» directly to it. Instead, we specify the «message» from the operation parameter.

Scenario 1: A *publish* operation containing a primitive type message. The message parameter must be stereotyped with «message».

```
«publish» private abstract String informSupplier(«message»String
aMaterial);
```

Listing 7- 2119 Published operation with a standard message datatype

Scenario 2: A *publish* operation containing a user-defined message. The message object must contain «message».

```
«publish» private abstract String informSupplier(Material aMaterial);
«message» class Material {
}
```

Listing 7- 22 Published operation with a message object

Scenario 3: A *publish* operation that expects to receive a result from the message consumer. In this case, there is no change to the *publish* operation. However, developers must define a «callback» operation that will be invoked asynchronously when the message consumer replies.

There are 3 stereotype attributes in the *publish* stereotype. The only required attribute is a channel name. If the modeler does not specify a channel type, the default “topic” is

used. If the modeler does not specify a message acknowledgement policy, the default "SyncWithServer" is used. There attributes reflect how a publish transformer generates code. The publish transformer generates code to publish a message by obtaining a message object from a static method, *MessageHelper.createMessage(Session session, Object message) : Message* provided by a message transformer (described later).

Notice that the *informSupplier* operation returns a string. This string is a unique identifier for the message sent to the channel and is used when the message producer needs to get results back from the message consumer. See the «callback» section below for more details.

Scenario 1a: A *publish* operation to a *topic* with a channel name.

```
«publish {channel="c1"}» private abstract String
   informSupplier(«message»String aMaterial);
```

Listing 7- 23 Channel name

will be transformed to:

```
private String informSupplier(String aMaterial) {
   Session session = _getMessageCoonection().createSession(false,
                  Session.AUTO_ACKNOWLEDGE);
   Topic topic = _getMessageContext().lookup("c1");
   MessageProducer producer = session.createProducer(topic);
   Message message = MessageHelper.createMessage(session, aMaterial);
   producer.publish(message);
}
```

Listing 7- 24 Generated code for channel

Scenario 1b: A *publish* operation to a *queue* with a channel name.

```
«publish {channel="c1",channelType="queue"}» private abstract void
   informSupplier(«message»String aMaterial);
```

Listing 7- 25 Publish with a channel name

will be transformed to:

```
private void informSupplier(String aMaterial) {
    Session session = _getMessageCoonection().createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
    Queue queue = _getMessageContext().lookup("c1");
    MessageProducer producer = session.createProducer((Destination)queue);
    Message message = MessageHelper.createMessage(session, aMaterial);
    producer.publish(message);
}
```

Listing 7- 26 Generated code of the publish operation

The transformer substitutes channel (topic/queue) lookup string with the channel name. The acknowledgement policy {ack} is used during the *createSession()*.

Scenario 2: A publish operation containing a user-define message. The message object must contain «message».

The generated code is identical to scenario 1a and 1b since the *MessageHelper* class is responsible for wrapping the user defined object into a JMS message.

Scenario 3: A *publish* operation that expects to receive a result from the message consumer.

The generated code creates a temporary channel that accepts a result message from the message consumer. In JMS, the temporary channel is passed to the consumer via the *setJMSReplyTo()* operation.

```
class Inventory implements MessageListner {
    …
    private String informSupplier(String aMaterial) {
           Session session = _getMessageCoonection().createSession(false,
                           Session.AUTO_ACKNOWLEDGE);
```

```
            Topic topic = _getMessageContext().lookup("c1");
            MessageProducer producer = session.createProducer(topic);
            Message message = MessageHelper.createMessage(session, aMaterial);

            Topic returnTopic = session.createTemporaryTopic();
            Message.setJMSReplyTo(returnTopic);

            producer.publish(message);
    }
}
```

Listing 7- 27 Generated code of the publish operation using JMS

«subscribe»

A subscribe stereotype applies to a message consumer operation that receives messages from a producer. Similar to the publish stereotype, this operation must contain one parameter that is a message object. The message object can be a user defined object or a primitive data type. If it is a user defined object, the user defined class must have «message». If it is a primitive data type such as UML::String, we cannot put «message» directly to it. Instead, we specify the «message» from the operation parameter. Since JMS requires a subscribe operation name to be *onMessage(Message message)*, the subscribe transformer must wrap the subscribe operation into this *onMessage()*. A subscribe transformer injects a subscriber object for the topic/queue in the class constructor.

There are 2 stereotype attributes. The only required attribute is a channel name. If the modeler does not specify a channel type, the default "topic" is used. Otherwise the subscribe method listens to a queue. The subscribe transformer generates code to accept messages by obtaining message object from a static method, *MessageHelper.createMessage(Session session, Object message) : Message* provided by a message transformer (described later).

Scenario 1: A *subscribe* operation containing a primitive type message. The message parameter must be stereotyped with «message».

```
      «subscribe {channel=”c1”}» private void receiveOrder(«message»String
aMaterial) {
            …
      }
```

Listing 7- 28 Subscribe operation

will be transformed to:

```
   class Supplier {
      private Session _session;
      public Supplier() {
            _initialize();
      }
      private void _initialize() {
            _getContext();
            _getMessageConnection();
            _subscribe();
      }
      private Session _getMessageSession() {
            return _session;
      }
      private void _subscribe() {
            session = _getMessageCoonection().createSession(false,
                  Session.AUTO_ACKNOWLEDGE);
            Topic topic = _getMessageContext().lookup(“c1”);
            session.createSubscriber(topic);
      }
      private void onMessage(Message message) {
            String aMaterial = MessageHelper.convertMessage(session, message);
            receiveOrder(aMaterial);
      }
      private void receiveOrder(String aMaterial) {
            …
      }
   }
```

Listing 7- 29 Generated code of the subscribe operation

Scenario 2: A *subscribe* operation containing a user-define message. The message object must contain «message».

```
«subscribe {channel=”c1”}» private void receiveOrder(Material aMaterial)
{
	…
}
«message» class Material {
}
```

Listing 7- 30 Subscribe operation with a message object

The subscribe transformer will generate identical code to the scenario 1 since it delegates the message conversion to *MessageHelper.convertMessage()* generated by the message transformer.

Scenario 3: A *subscribe* operation that returns a result object. A message consumer operation can return a result object back to the message producer. In this situation, the consumer becomes a message producer and it publishes the result message back to the original producer. If the return object is a primitive type, it must have «message». If it is a classifier, its classifier definition must have «message».

```
	«subscribe {channel=”c1”}» private «message» String receiveOrder(Material
aMaterial) {
		…
	}
```

Listing 7- 31 Subscribe operation with a result object

The JMS implementation is:

```
	private void onMessage(Message message) {
		String aMaterial = MessageHelper.convertMessage(session, message);
		String result = receiveOrder(aMaterial);
		Topic returnTopic = message.getJMSReplyTo():
		Publisher publisher = _getMessageSession()
			.createPublisher(returnTopic);
```

```
        publisher.publish(returnTopic);
    }
    private void receiveOrder(String aMaterial) {
        …
    }
```

Listing 7- 32 JMS implementation that publishes a return message

«message»

A message is a serialized object that is sent from a producer to one or more consumers. It is therefore a kind of persistent objects that can re-use APIs (_getSerializedString() and _getDeserializedString()) from the persistence service from chapter 3. This stereotype contains an ordered set of message converters. Each converter implements *MessageConverter* interface (Listing 7- 11) and is applied in the order. In the end, the _getSerializedString() is called to create a string-based message. At the message consumer, the string message is deserialized and converted back in the reverse order to the original message object and passed back to the *subscribe* operation.

If the message's _getSerializedString() returns null, it indicates that the message cannot be serialized by the persistence middleware. This is a case when a persistence implementation such as SQL does not store objects in a serialized format. A transformer will throw a transformation exception and the developer must specifically choose a compatible persistence method for the message.

«callback»

Sometimes a message producer wants to receive results after it sends a message. The messaging profile defines an operation that is invoked when a message consumer returns results as a callback operation. A callback operation has a callback stereotype and must contain a channel name to indicate which channel it waits for a result. A callback is a special kind of a *subscribe* operation that subscribes to a named channel but its sole

purpose is to receive results back from a message consumer. If the message consumer operation does not return a value, developers do not need to define a callback operation.

Unlike the subscribe operation, the callback operation is defined at a message producer side and does not return a value. Similar to the *subscribe* stereotype, this operation must contain one parameter that is a message object. The message object can be a user defined object or a primitive data type. If it is a user defined object, the user defined class must have «message». If it is a primitive data type such as UML::String, we cannot put «message» directly to it. Instead, we specify the «message» from the operation parameter.

Even though messaging profile can be extended to support a synchronous request/response, it is discouraged. Developers should use the synchronous method invocation from the object distribution service instead.

The callback stereotype contains one stereotype attribute, a channel name. This channel name must match with the channel name from a *publish* operation.

Scenario 1: A callback operation containing a primitive type message. The message parameter must be stereotyped with «message».

```
      «callback {channel="c1"}» private void orderResult(«message»String
result) {
            …
      }
```

Listing 7- 33 Callback operation

will be transformed to:

```
   class Supplier implements MessageListener {
      private void onMessage(Message message) {
            String returnResult = MessageHelper.convertMessage(session,
                  message);
            orderResult(returnResult);
      }
   }
```

Listing 7- 34 Generated result of the callback operation

Scenario 2: A callback operation containing a user-defined message. The message object must contain «message».

```
	«callback {channel=”c1”}» private void orderResult(OrderAcknowledgement
anAcknowledgement) { … }
	«message» class OrderAcknowledgement {
	}
```

Listing 7- 35 Callback operation with a message object

The callback transformer will generate identical code to the scenario 1 since it delegates the message conversion to *MessageHelper.convertMessage()* generated by the message transformer.

7.6 Messaging Transformation

The previous section shows how a model that uses the messaging profile is transformed into a particular implementation. This section systematically defines model transformation into two steps; the marked PIM-to-annotated PIM and the annotated PIM-to-PSM transformations. It also describes how each stereotype transformer is implemented and shows that some of them reuse services provided by other profiles.

7.6.1 The marked PIM to annotated PIM transformation

When a stereotype is applied to a model element in a PIM, it enhances the model element with additional properties. For example, a «producer» adds two public operations; _getMessageConnection() and _getMessageContext() to a producer class. Model developers can call these two operations without knowing how these operations are implemented. Another example is the *MessageConverter* interface that can be used to convert and deconvert message objects. This interface is available when the Messaging profile is imported.

The Mercator PIM-to-PIM transformation is a layer that enhanced the marked PIM with additional constructs such as operations, classifiers and default stereotype attributes.

Since the Mercator transformation is triggered by stereotype, each stereotype transformer is responsible to add additional constructs to the model elements. The Table 7- 2 below shows constructs that are added to a model element when a stereotype is applied.

Name	*Applies To*	*Constructs*
«profileImport» {name=Messaging}	Model	Allows models to use messaging service.
«producer» {init="pre" | "lazy"}	Classifier c	_getMessageConnection() : Connection _getMessageContext() : Context
«consumer»	Classifier c	_getMessageConnection() : Connection _getMessageContext() : Context
«message»	Classifier or Parameter	_getSerializedString() : String _getDeserializedString(in String s) : Object

Table 7- 2 Constructs introduced to model elements when stereotypes are applied.

The messaging profile uses services from Naming and Persistence profiles. Three stereotypes; «publish», «subscribe» and «callback» contains a channel name. They are defined as sub-stereotypes of the «singleton» from the naming profile and the *channel* attribute is a redefinition of the *name* attribute. However, these three stereotypes cannot directly reuse the «singleton» transformer because they also contain additional attributes. In fact, these three transformers subclass from the «singleton» transformer and add transformation code for the additional attributes.

The «message» is a sub-stereotype of the «persistence» where the *id* attribute is autogenerated. Since the «persistence» defines the *_getSerializedString()* and *_getDeserializedString()*, the message object can use them to convert between the object and its serialized textual representation.

7.6.2 The annotated PIM to PSM transformation

A messaging profile defines a set of PSM transformation choices. Each choice contains a messaging middleware name, its messaging transformer and a middleware specific annotation file.

Choice	*Transformer class in org.mercator.profile.messaging package*	*Annotation file*
JMS1.0	JMS10Transformer	jms10.annotation.xml
JMS1.1	JMS11Transformer	jms11.annotation.xml
EJB_MDB	MDBTransformer	mdb.annotation.xml
AxisWebService	AxisWSTransformer	axis.annotation.xml

Table 7- 3 PSM Transformation choices for the messaging profile.

Since each middleware specific transformer has its own annotation file, the annotation file contains different parameters specific for each concrete middleware. Therefore it is possible to switch back and forth between each choice. When a choice is made, the transformer checks whether its annotation file exists. If it does, the transformer will use parameters from the annotation file; otherwise, it will create a new annotation file with default properties. Subsequent transformation of the same choice will reuse from the existing annotation file.

7.7 Summary

Messaging enables enterprise applications to exchange data asynchronously. However, different messaging middleware defines APIs, messaging policy and message structure differently. This chapter introduces a middleware independent messaging profile that allows developers to define message payload, producer, consumer and operations that do not depend on a specific middleware. The profile consists of 6 stereotypes that a model can use to define which model elements are message producers, consumers as well as which operations publish and subscribe to message objects.

Using messaging profile allows developers to define and use asynchronous message exchange without knowing implementation details. Supporting new messaging middleware can be done by creating a new middleware specific transformation and defining the new transformer to the messaging profile.

Chapter 8 Implementation

8.1 Introduction

This chapter describes the Mercator workbench that implements the Mercator framework and provides an environment for modelers to model and annotate PIMs and generate PSMs and subsequent source code. The workbench contains six components.

1. *The model representation* describes how models are stored in textual formats. The workbench reads textual models and represents them in various forms; abstract syntax trees, tables, UML diagrams or source code.
2. *The transformation component* defines interfaces that model compilers must implement to create transformers that translate model elements from one modeling language to another. Each transformer is associated with a stereotype. The transformation process visits stereotyped model elements in an input model and executes transformers associated with those stereotypes.
3. *The profile definition* groups a set of related stereotypes into object services and provides default annotation templates that allow modelers to customize the transformation.
4. *The factory repository* keeps factory objects that are responsible for constructing, storing and loading model elements. PIM, MPIM and PSM each has its own factory object. Model compiler developers can define new modeling languages by adding a factory object into this repository.
5. *The user interface* is the interactive development environment for modelers to construct, manipulate and transform PIMs into PSMs and subsequent source code.
6. *The model import utility* is an Eclipse plugin that converts a Java Eclipse project into the Mercator PSM.

8.2 Model Representation

A model is represented by an AST tree. Each AST node follows the extended eMOF data structure in the Chapter 2 and can be used to describe different modeling languages

such as a PIM, an MPIM and a Java PSM. Each language contains many subclasses of the AST node that implements the IAstNode interface defined in appendix B.

The IAstNode interface provides APIs to access node properties, relationships, stereotypes and transformation exceptions. Modeling language designers subclass the AST node to define a model grammar for each language. For example, a Java PSM grammar contains AST node definitions for the Java package, type, class, interface, field, method and so on while a UML PIM grammar contains ones for the UML package, type, class, interface, attribute, operation and so on. The Mercator tool uses interfaces defined in the IAstNode to inspect and transform model instances from different modeling languages.

8.3 Transformation Component

The transformation component contains validators and transformers. The validators verify that stereotyped model elements meet constraints defined by the stereotype. Otherwise, it will put validation exceptions in the model element. The validators are used to check the consistent state of the model. It is possible for a model to be partially consistence but the model transformation will not be successful until all nodes are valid. The transformers generate output model elements. Validator implements the *IValidator* interface while transformers implement the *ITransformer* interface. Both interfaces use the Visitor pattern [GHJV95].

```
package org.mercator;
…
public interface ITransformer extends IVisitor {
    public IAstNode transform(IAstNode input, Object arg);
    public IAstNode preTransform(IAstNode rootNode, Object arg);
    public IAstNode postTransform(IAstNode rootNode, Object arg);

    public void setDefaultAnnotationFileName(String fileName);
    public String getDefaultAnnotationFileName();

    public void setLevel(int level);
    public int getLevel();
```

```
}

package org.mercator;
…
public interface IValidator extends IVisitor, Observable {
    public List<AbstractException> validate(IAstNode root);
}
```

Listing 8 - 1 ITransformer and IValidator interface

8.4 Profile Definition

Mercator uses XML to define a profile. A profile consists of the name of the service, a list of stereotypes and a default annotation file. Each stereotype has a name and a model element for which it is marked and a list of stereotype transformers. If modelers don't choose a transformer choice of the stereotype, a default transformer will be used. A transformer definition has a name, an ITransformer class name and a default annotation file. The annotation file contains default parameters used by stereotypes in the profile. Whenever, a profile is imported into a model, the default annotation file will be copied into an annotation file that can be modified. Listing 8 - 2 shows a profile definition for the persistence service.

```
<profile name="persistence"
  defaultAnnotation="default.persistence.annotation.xml">
  <stereotype name="persistence" forNode="pim::class"
    defaultTransformer="persistence">
     <transformer name="persistence"
    className="org.mercator.transformer.PersistenceTransformer"
    defaultAnnotation="default.persistence.annotation.xml"/>
  </stereotype>
  <stereotype name="id" forNode="pim::property"/>
  <stereotype name="persistence" forNode="psm::class"
    defaultTransformer="hibernate">
     <transformer name="hibernate"
    className="org.mercator.transformer.HibernateTransformer"
    defaultAnnotation="default.hibernate.annotation.xml"/>
     <transformer name="EJB-CMP"
    className="org.mercator.transformer.HibernateTransformer"
    defaultAnnotation="default.hibernate.annotation.xml"/>
```

```
    </stereotype>
    <stereotype name="id" forNode="psm::field"/>
  …
  </profile>
```

Listing 8 - 2 A persistence profile definition, persistence.profile.xml

The Mercator configuration file, *mercator.xml*, contains all profile names and their profile definition files.

```
    <profiles>
      <profile name="PIM" file="pim.profile.xml"/>
      <profile name="PSM" file="psm.profile.xml"/>
      <profile name="persistence" file="persistence.profile.xml"/>
      <profile name="unitofwork" file="unitofwork.profile.xml"/>
      <profile name="messaging" file="messaging.profile.xml"/>
      <profile name="naming" file="naming.profile.xml"/>
      <profile name="distribution" file="distribution.profile.xml"/>
    </profiles>
```

Listing 8 - 3 A profiles definition section in mercator.xml

An annotation file is an XML document with the annotation namespace. It contains information about the transformation method for each service. The method can be specified globally with the *forNode="*"* attribute or specifically for each stereotyped node element. For example, a model can choose Hibernate as a global persistence method and use the XML persister for the examples.Configuration class.

```
  <?xml version="1.0" encoding="UTF-8"?>
  <annotation:annotation xmlns:annotation="http://mercator.org/annotation">
    <annotation:transformerChoice forNode="*" forModel="pim"
stereotype="persistence::persistence" transformerName="persistence" />
    <annotation:transformerChoice forNode="*" forModel="psm"
stereotype="persistence::persistence" transformerName="hibernate" />
    <annotation:transformerChoice forNode="examples.Configuration"
forModel="psm" stereotype="persistence::persistence" transformerName="XML" />
    <annotation:id autoGeneratedIfNotSpecified="true"/>
  …
  </annotation:annotation>
```

Listing 8 - 4 An annotation for the persistence profile, default.persistence.annotation.xml

8.5 Factory Repository

The workbench uses factory objects for constructing, storing and loading model elements which are subclasses of IASTNode for a given language. Model compiler developers define a new modeling language, create a factory object for each language and add the language factory to the repository. A factory object is referred by the framework by its namespace. When the Mercator workbench opens a model file, it will check the model namespace and delegate the model loading to the factory. If there is no factory for the namespace, the tool will generate an error. Model compiler developers add a new factory definition in the factories section of the Mercator configuration file, *mercator.xml*.

```
     <factories>
       <factory className="org.mercator.repository.RepositoryFactory"
namespace="http://mercator.org/"/>
       <factory className="org.mercator.ast.PimFactory"
namespace="http://mercator.org/pim"/>
       <factory className="org.mercator.javaAst.JavaPsmFactory"
namespace="http://mercator.org/psm/java"/>
       <factory className="org.mercator.annotation.AnnotationFactory"
namespace="http://mercator.org/annotation"/>
     </factories>
```

Listing 8 - 5 A factories definition section in mercator.xml

Each factory object extends an *AbstractFactory* class. It must be able to create, look up and remove model elements in a model. Since the IASTNode interface contains methods that serialize the node from and into XML string, the factory can traverse a root node and generates a serialized XML file from a model and vice versa. See listing D-2 in Appendix B for the AbstractFactory interface definition.

8.6 User Interface

The Mercator workbench provides a graphical user interface for modelers to manipulate and transform models.

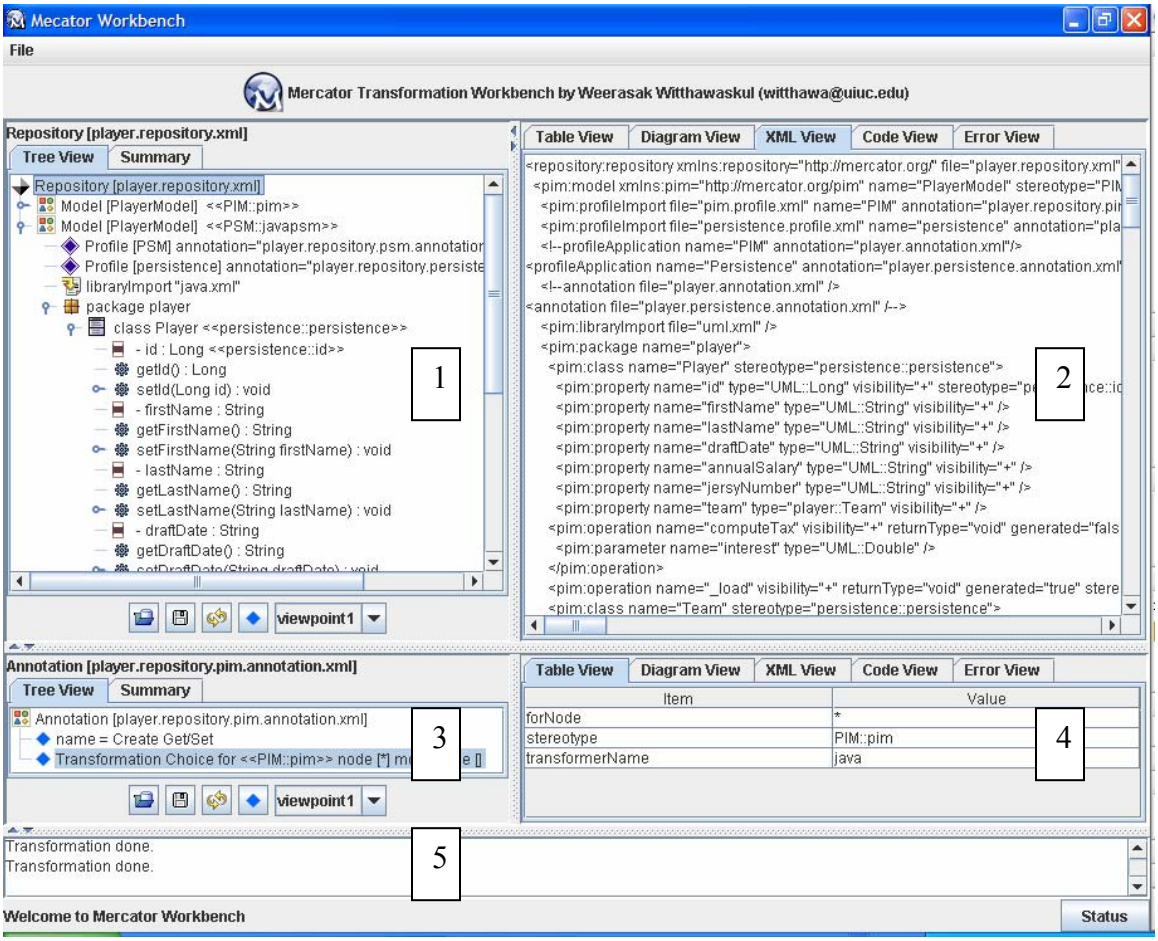


Figure 8 - 1 Mercator Workbench UI

The tool consists of five panes. The first pane (1) shows the tree view of the model repository. The second pane (2) allows modelers to view and modify current model element selected from the first pane. Modelers select a model element in (1) and edit it in (2). There are different ways to display selected model element, for example, in table view, diagram view, XML, embedded code or transformation error messages if exists. The third pane (3) displays an annotation tree of the last selected profile. Properties of the selected annotation node from the third pane can be modified in the fourth pane (4). When modelers select a profile in (1), associated annotation file will appear in (3). If modelers select an annotation element in (3), they can edit annotation properties in (4). The fifth pane (5) shows logging and debug messages.

A modeler uses the tool GUI to load, mark and transform models. First he opens a model file. This file contains two classes; Player and Team. He wants to add persistence service to this model so that objects created from these classes can be stored in a database. He right click on the root model node to bring up a pop up menu and select Import profile > persistence (Figure 8 - 2).

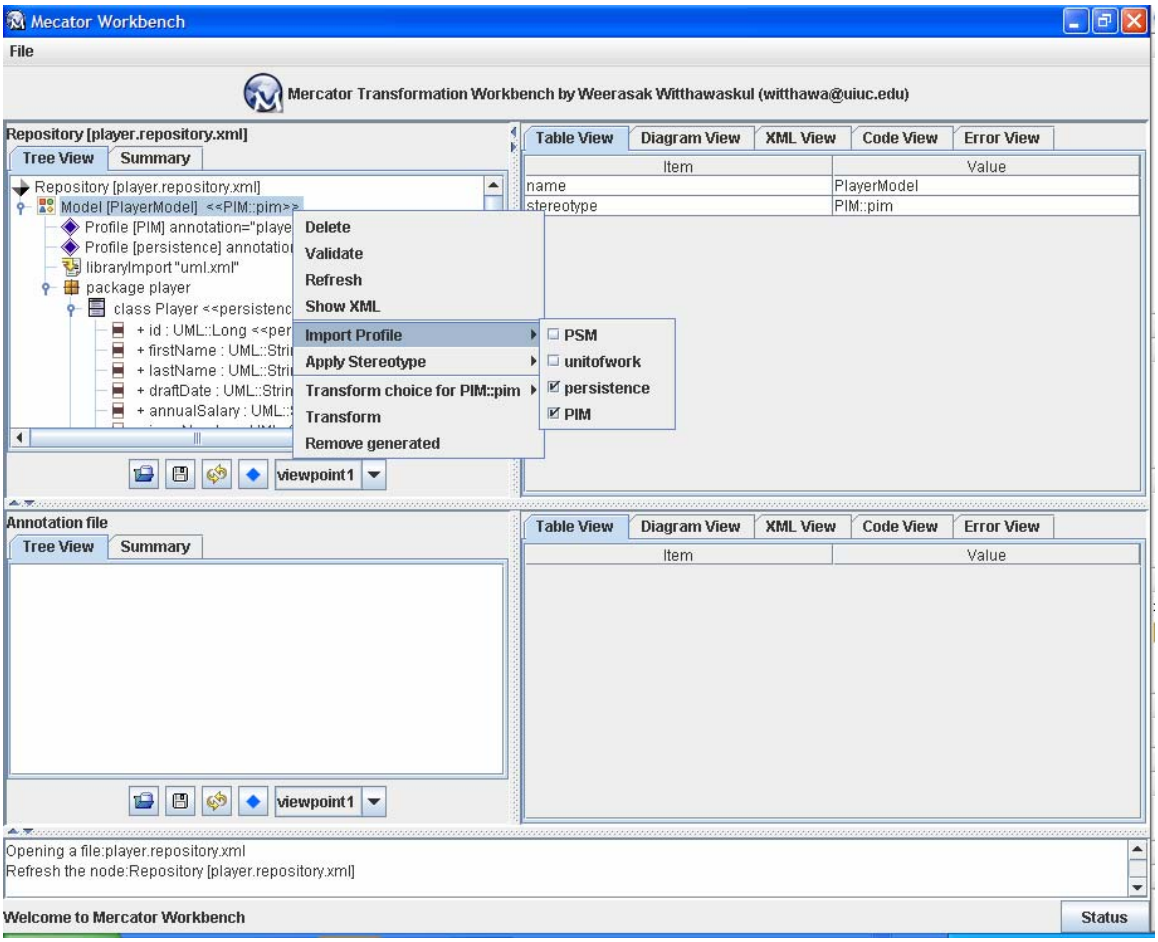


Figure 8 - 2 The Mercator tool GUI

Once the profile is imported, he can now use persistence stereotypes to mark the Player and Team classes. He also marks class an attribute identifier with «id» to indicate the persistence id for each class. After he finishes, he selects the model root again to bring up a popup menu and select Transform (Figure 8 - 3).

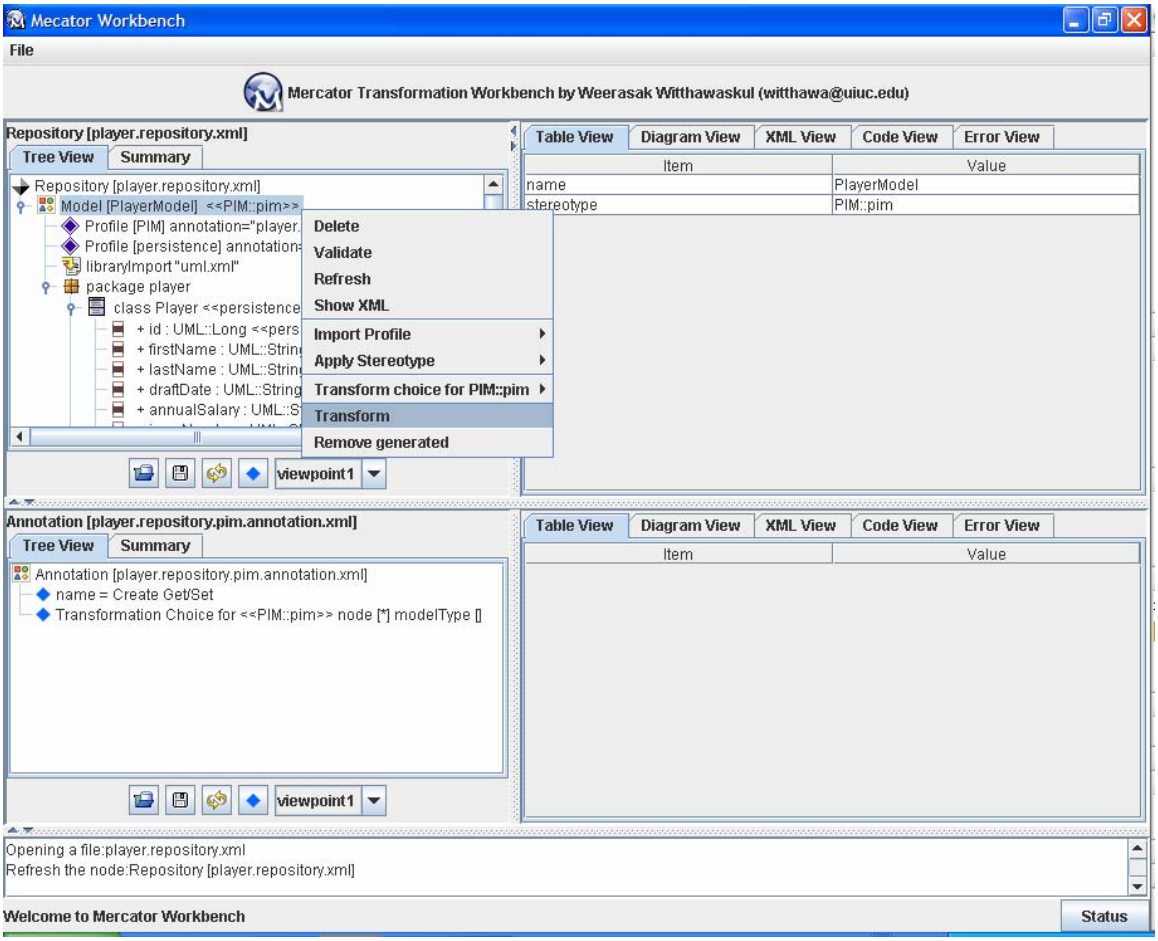


Figure 8 - 3 Transform menu

After the transform menu is selected, Mercator looks up a transformer for the model root. Since the model root has a «pim», it invokes the PIM transformer. The PIM transformer visits each model node in the tree and collects persistence transformers that associate with the «persistence» and «id» into a transformer queue. After it visits all the nodes, it invokes transformers in the queue. It then marks the input model as an MPIM. In the next pass, Mercator invokes the MPIM transformer over the MPIM and creates a PSM.

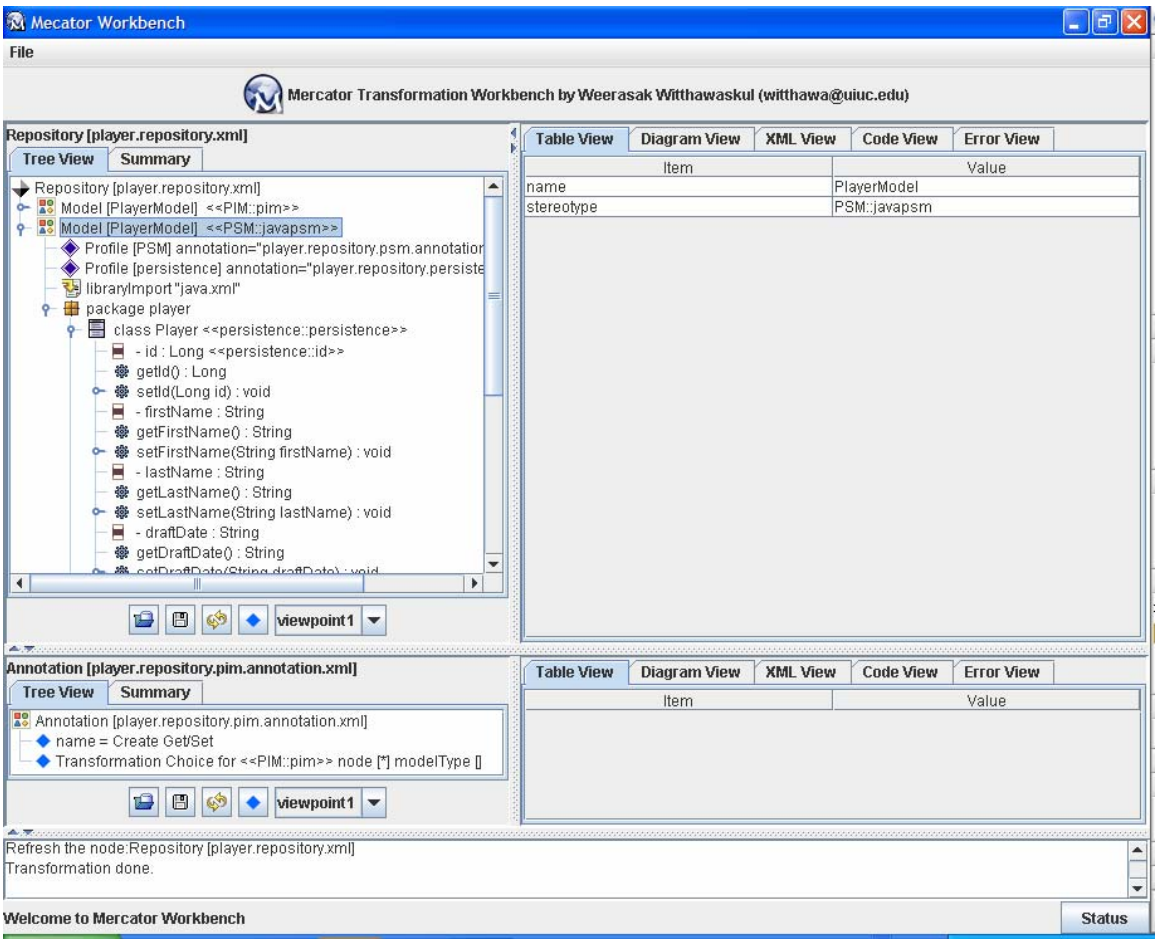


Figure 8 - 4 The PSM is generated by the PIM tree transformer

Figure 8 - 4 above shows the result of the transformation. The modeler expands the PSM model tree to see the result. The PSM output classes contain getter/setter operations as well as persistence related methods. The modeler can view generated source code for each model element from the Code View as in Figure 8 - 5.

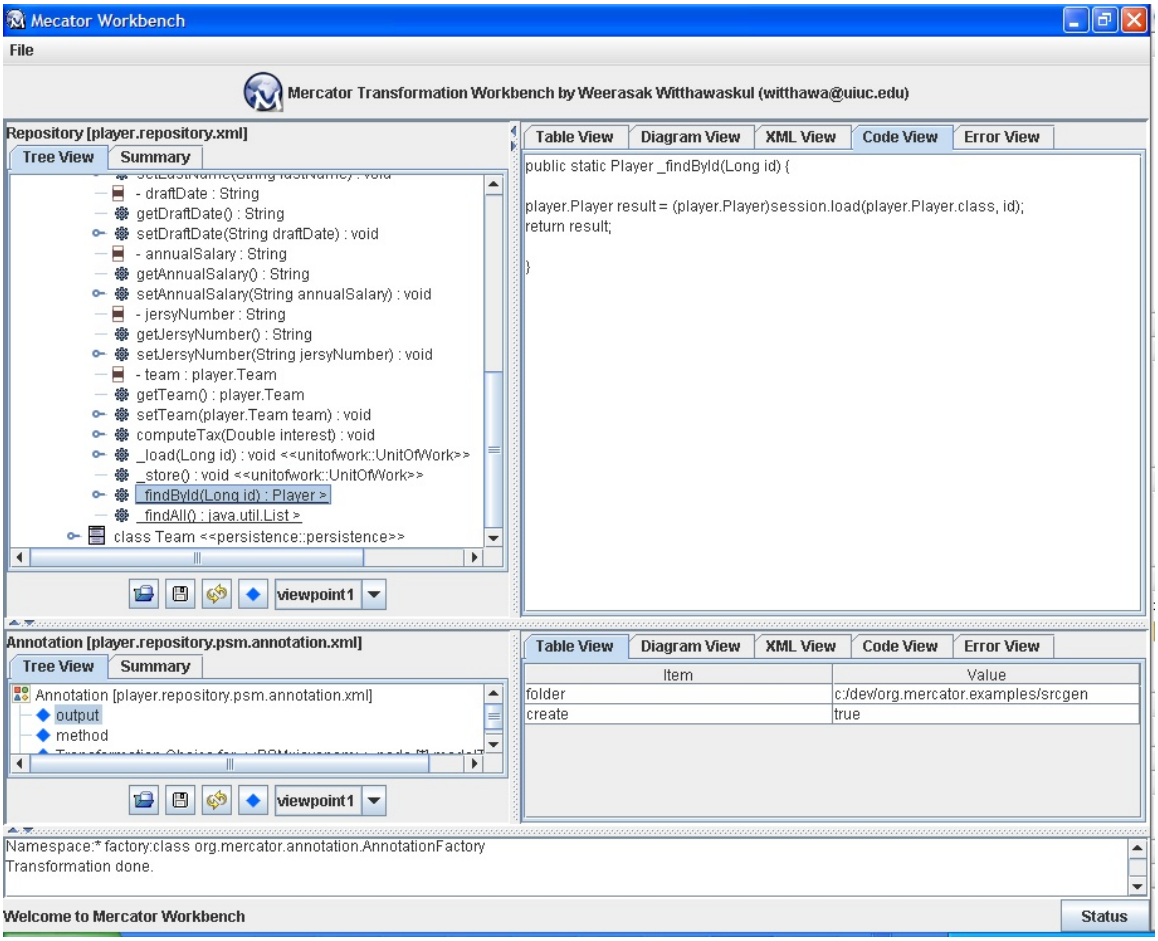


Figure 8 - 5 Generated code for the finder method *_findById()*

Since the PSM is just another model, the modeler select the Transform menu over the PSM to generate source code. The modeler can specify where the source code is generated by changing a PSM annotation parameter 'folder' (the bottom right of the Figure 8 - 5). Mercator looks up and invokes a PSM transformer which produces Java source code. The modeler can then compile and test the generated code. If he wants to change the model, he can go back and change the PIM and retransform.

8.7 Model Import Utility

This utility provides a Java source code to the Mercator model file. It is an Eclipse plugin that converts an Eclipse project into a Mercator model file by traversing Eclipse AST nodes and emitting Mercator model elements. The result model is an unmarked Java PSM. Modelers still need to clean up the result model to remove dependencies to

specific middleware and use the GUI to import profiles and mark model elements with stereotypes. Alternatively, modelers can choose to build a model from scratch which is more suitable for small systems.

8.8 Evaluations

We will use a simple class model from Figure 8 - 6 to measure transformation performance. This example uses the persistence service and uses the Hibernate persistence method.

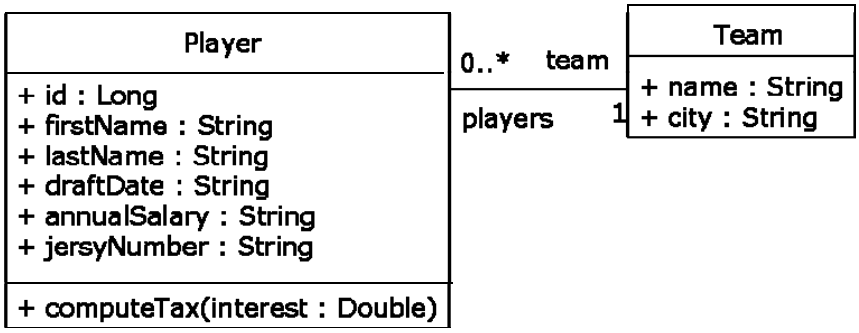


Figure 8 - 6 Player - Team class diagram

In the Mercator model tree explorer, the input model imports the default Mercator profile and marks the model with «PIM». Next it imports the persistence profile and the Player class is marked with «persistence» to indicate that the instances of this class must be persistent. The Player class already has an id attribute so we mark it with «id» as shown in Figure 8 - 7.

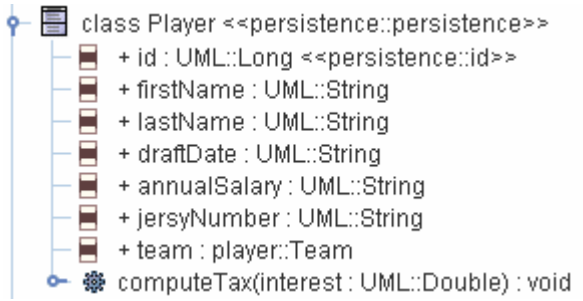


Figure 8 - 7 Player class with «persistence»

The Team class is also marked with «persistence». However, the team does not have an id field. Since the default id generation policy automatically generates a surrogate id if one does not exist, a persistence transformer will add an id attribute with «id».

When the developer transforms the model, the tool will find a PIM transformer associated with the «PIM» root node. The PIM tree transformer iterates over all model elements and finds a «persistence» transformer for the Player class. The persistence transformer validates the persistence class and creates method signatures for persistence APIs. Four of them are static methods while two are regular methods. Figure 8 - 8 shows the result of the persistence transformation. The result is an MPIM which is a model with persistence methods but does not yet contain method implementations.

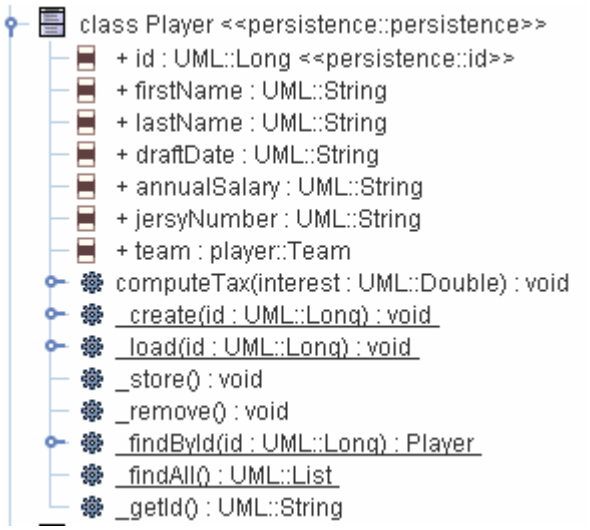


Figure 8 - 8 Player class with generated persistence APIs

Next, the developer chooses a Hibernate persistence library as the persistence method. The Mercator tool looks up the Hibernate persistence transformer and invokes it. The transformer is responsible for code implementation of each persistence API. At the end, the tool invokes the PIM-to-JavaPSM transformer that creates Java classes from PIM classes, adds getter and setter methods for each non-transient class field and maps UML data types to Java types. The result PSM is shown in Figure 8 - 9. The figure does not show method body and Hibernate mapping and configuration artifacts that the transformer generates. The resulting model is marked with «PSM».

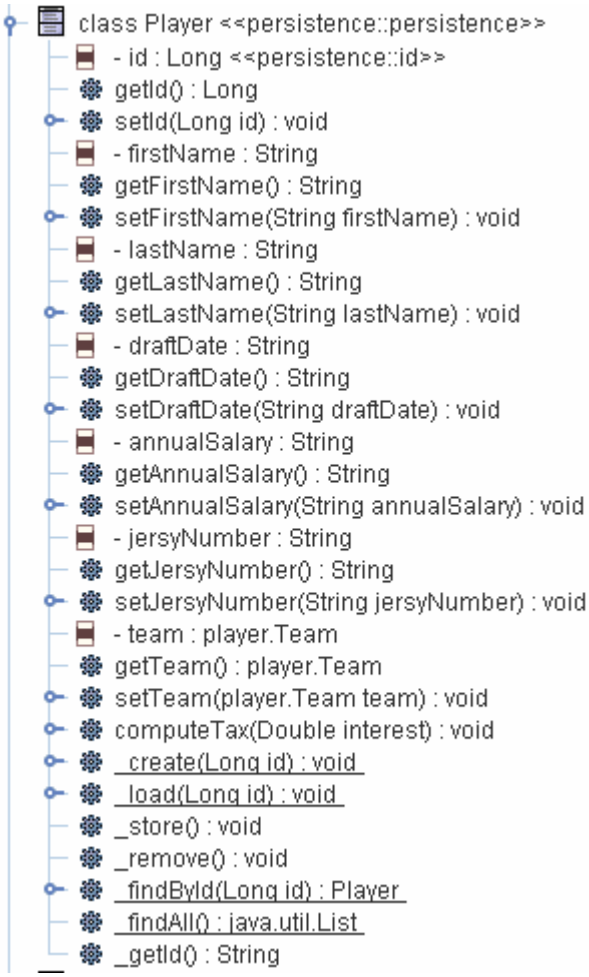


Figure 8 - 9 Player java class implementation

Once the PSM is created and marked with «PSM», the transformation tool finds a PSM-to-Code transformer associated with this «PSM» from the transformation registry and invokes it. The result contains source files generated into an output directory where all Java class elements are generated into Java class files and artifact elements are generated into files. Listing 8 - 6 below shows the method body of the `Player#_findById()` method generated by the Hibernate persistence transformer.

```
public static Player _findById(Long id) {

Session session = HibernateUtil.currentSession();
Transaction tx = session.beginTransaction();
player.Player result = (player.Player)session.load(player.Player.class, id);
tx.commit();
HibernateUtil.closeSession();
return result;

}
```

Listing 8 - 6 _findById() method body

Since this example contains only 2 classes in one package, we created bigger examples to measure the time and space performance. There are 4 input model files. Each model file contains different number of packages, classes and class features that include class attributes and operations. All transformations are tested in a 1.83Ghz Intel Core Duo CPU with 1.5GB RAM and a 5,400rpm hard drive.

In the PIM section in the table, each model is loaded into the Mercator workbench and the load times are shown in the PIM section. It is note that the 3rd model file load time is slightly faster than a smaller 2nd model file. This may result from a variance from timing and file cache.

The second section is the MPIM which is the PIM with APIs introduced by a persistence profile. Notice that the class features has been expanded between 1.8 and 2.4 times. The transformation time per input class element is highest when the input model elements are small (12.8ms) and lower when the model elements are larger which are 1.85ms, 1.76ms and 1.67ms respectively.

The third section contains results from the MPIM-to-PSM transformation with Hibernate as the persistence method. Notice that there is one additional package that contains the HibernateUtil helper class and there are artifacts that are hibernate ORM definition for each class and the main hibernate configuration file. The transformation takes less than 1 second for a 2-class example and 3.4 seconds for a 250-class example.

The final section shows results from the model-to-code generation. The number of files generated is double the number of the input classes due to the ORM definition file for each class. The total generated file sizes compared with the input model files increase between 2.6 and 3.3 times.

Model File	Player 1p2c	Player 10p40c	Player 10p50c	Player 50p250c
PIM				
#packages	1	10	10	50
#classes	2	40	50	250
#class features	11	220	275	1375
Model file size (KB)	3	38	47	232
Load time (ms)	312	625	610	1079
MPIM				
#packages	1	10	10	50
#classes	2	40	50	250
#class features	26	400	650	3250
Transform time (ms)	141	406	485	2297
PSM				
#packages	2	11	11	51
#classes	3	41	51	251
#class features	30	404	654	3254
#artifacts	3	41	51	251
Transform time (ms)	625	985	1266	3391
Generated files				
#files	6	82	102	502
#lines	300	4556	6026	28,076
#uncommented, non-blank lines	275	4265	5665	26,315
#bytes (KB)	8	114	155	703
Average lines per file	50	55	59	55
Average bytes per file	1355	1395	1518	1400

Table 8 - 1 Transformation results

Figure 8 - 10 shows a graph comparing result model sizes in KB, load time, and transformation performance in milliseconds during the PIM-to-MPIM transformation (PIMT) and the PSM-to-Code transformation (PSMT) of the four input models. Notice that the transformation time scales linearly to the number of input class elements. Since

Mercator does not contain any optimization, we believe that a total load and transformation time for a typical, large system that contains several hundreds classes will not take more than 10 seconds.

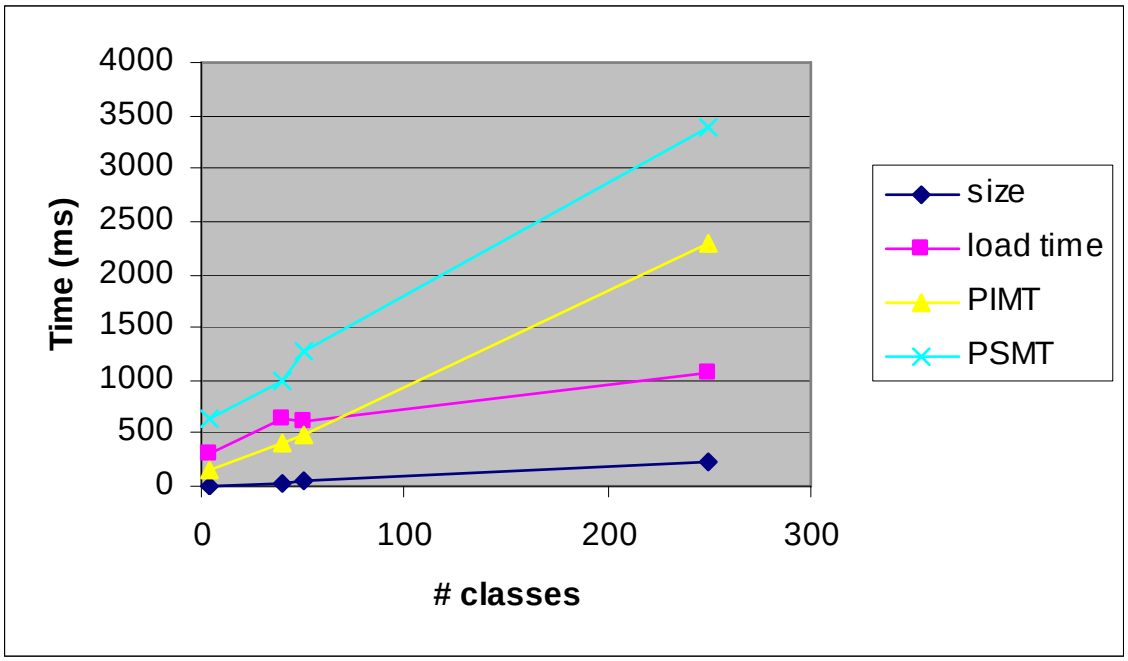


Figure 8 - 10 Transformation time and model size

We do not have published performance data of other transformation systems to compare. In fact, model transformation research is still an early stage where the focus is not about performance. Since each modeling languages/frameworks use different model formats and transformation methods, it's probably unfair to compare non-optimized transformation systems. An author of one declarative model transformation admitted that transformation performance is the worst feature of declarative transformation languages. We believe that our direct model manipulation approach used in our framework has a better performance and can scale linearly with the number of input class elements as indicated from results in our prototype.

8.9 Lessons learned

We have learned tremendously during the design and especially from the prototype implementation. We understand why there are many meta-modeling tools in the market

as they were designed for different needs. It has been difficult to base a design on a fast-changing tool and we finally implemented our own so that a model representation can be greatly simplified and still support key concepts such as namespace, UML profiles and viewpoints.

We discovered that by delegating model element management to a factory object, it is easier to support new modeling languages. The Mercator framework uses the extended eMOF to represent an object oriented system and use pluggable factories to create, find and remove model elements. For example, to create a class method, a PIM factory creates a model element called 'operation' while a Java factory creates a model element called 'method'. A more complicated example is when a persistence transformer takes an input persistence class in a PIM called 'StockMarket' and generates two elements; a persistence interface 'StockMarket' and a persistent implementation class 'StockMarketImpl'. At a PIM level, modelers only see the persistence StockMarket class and do not have to know how the actual implementations are generated.

We designed two model-to-model transformers that model compiler developers can subclass to write new transformers; a tree transformer and a node transformer. At first we believed that both transformers should be used in the same frequency. However, we found out that tree transformers are only used during a PIM-to-PSM and a PSM-to-code where all elements in a model tree must be visited. Stereotype transformers, on the other hand, are all node transformers. Since the framework provides model manipulation APIs, a transformer has access to the stereotyped node element as well as an ability to traverse within the node subtree or to its parent. Therefore, very few transformers derive from the tree transformer and most of the others derive from a node transformer.

The Mercator workbench is a good learning experience by itself. The GUI was designed using the MVC pattern so that models and GUI are separate. The model is not aware of the GUI. Mouse clicks, node selections, menu selections, property edits are mapped into commands that apply to the model by a model manager. The result is that it has no

different to transform models from either the interactive workbench or automated test cases.

The Mercator framework allows model compiler developers to plug in factory objects to support different kinds of models. It also allows them to add new profiles, stereotypes and transformers to create new middleware independent services. Once the framework loads these profiles, a model can use the stereotypes defined in new profiles and the framework know which transformers to invoke to generate a PSM and subsequently source files. The framework provides flexibility needed to support new middleware implementation targets and future middleware independent services.

Chapter 9 Conclusions

9.1 Summary

A paradigm shift in software development from code-centric to model-centric requires precise definition and transformation of models. The abstraction of models that do not depend on middleware libraries allows software models to show intents better and makes them simpler to maintain because they contain less model elements and easier to migrate to different middleware as technology evolves.

By using stereotypes defined in each profile, modelers specify additional capabilities for model elements; some capabilities extend model elements with extra APIs that modelers can use without knowing how these APIs are implemented. Implementations of these APIs are generated by stereotype transformers that are associated with the stereotypes. Each transformer corresponds to a middleware library. Modelers can retarget the same middleware independent model to use different middleware by changing the transformer.

The Mercator model transformation framework provides a standard way to define object services as a set of profiles; each profile consists of related stereotypes. The framework allows model compiler developers to associate transformers for each stereotype and plugin transformers into the transformer registry. Model transformer developers use the framework model manipulation APIs to access model elements and annotation parameters. The framework provides tree transformers that iterate a model tree from a root node, collect transformers for model elements that have stereotypes and apply these transformers in an order defined by profile dependencies. There are 2 kinds of tree transformers; one is a model-to-model transformer from PIM to Java PSM and the other is a model-to-text transformer from Java PSM to Java source files.

9.2 Summary of Contributions

In summary, the main contributions of this dissertation are:

- A design of profiles for middleware independent services in the following areas:
    - Persistence
    - Naming
    - Unit of Work
    - Distribution
    - Messaging
- An object oriented, model transformation framework based on stereotype triggers. This framework provides a systematic way to create new service profile, add new transformers to existing profiles
- A model transformation prototype that contains at least two different implementations for each service.
- A simple XML-based model representation that supports UML class model and Java viewpoints. This representation extends from the OMG's eMOF specification to support UML profiles, validation and viewpoints.

9.3 Limitations

The approach used in this dissertation fits nicely with the vision of the OMG Model Driven Architecture. The framework does not solve all aspects of the platform independence. Its scope is applied to the middleware aspect. We believe that modeling software applications that do not depend on specific middleware libraries yields substantial benefits to model reuse and automated model and code generation. However, our approach has the following assumptions which are also limitations.

- Interoperability of modeling tools. By the time we designed the model transformation, there were many meta-modeling libraries [EMF] [MDR] [Eme05] to choose and they were evolving rapidly. We couldn't find public libraries that allowed us to easily add the UML profile concept as well as supporting relaxed consistent model elements. For example, a persistent class must have an identifier. We cannot mark a class id field before we mark the class persistence because an id field must be defined within a persistence class. In practice, we would like a flexibility that allows the class persistence and id

markings in any order, thus the model may be partially inconsistent. The validity check will be performed when modelers instruct the model transformation. Therefore, we chose to define our own model representation based on the eMOF and extended it to support profiles. We believe that we could create an import/export tool to generate model files for most commonly used formats. However, the model synchronization between formats is likely an ongoing issue as it usually is in the current model repository research.

- Lack of declarative transformation language. Model transformation is still an active research area. There are many approaches to transform models. The Mercator framework did not invent yet another transformation language but instead uses XML to define profiles and uses Java to create transformers. Even though the framework provides transformer class hierarchy and model manipulation APIs, model compiler developers must write Java code to implement transformers. It can be argued that imperative approach like this object oriented framework is easier to understand, has a better transformation performance, and more scalable by having transformers for each stereotype. Since other model transformation research do not yet publish a transformation performance and each implementation uses different representation, it's hard to make a direct comparison. The latest specification from the OMG Query, View and Transformation (QVT) is in a final adopted specification and will be publicly released some time in late 2006 where we start to see implementations in commercial products. However, the QVT focuses on a model-to-model transformation and does not yet address middleware independent modeling nor specifically put parameters in separate annotations.

- Language dependence. As we use Java to describe behaviors of class models, the models depend on the language. This limits the use of the framework to IT shops that commit to the Java platform. The UML superstructure specification [UML05] defines action semantics that allows for language independent behavior specifications. However, the specification only provides an abstract syntax

model of the action semantics and leaves concrete action language definitions to tool vendors. The idea is that any concrete action language that follows the abstract syntax specification will be able to describe model behaviors and even converted into code in different programming languages.

- Standards. MDA provides related specifications [MDA01] and encourage companies to use them. However, key specifications are still underway and some companies found the specifications inconsistence, lacks of reference implementation or compatibility suites. The framework has been designed from 2003 when these specifications were not finalized or were in request for proposal state. We believe that the object oriented framework is one of the practical approaches that realizes the MDA vision and can be used as one of the reference implementations.

9.3 Future Work

Even though the framework provides a solid groundwork for model transformation, it needs to be used and evolved. There are many possibilities to expand the framework to support more features. They are described below.

9.3.1 Model Versioning

Version control system is a well known method to manage source code and support team programming. Developers use a version control system to define software releases and be able to back track to versions that they are interested in. Version control also provides check-out/check-in methods so that developers in a team can check out and work on a part of the software and merge the changes back to a centralized repository. Model versioning is similar but instead of storing source and executable files, it keeps a repository of model files. A model versioning system can be used to track changes in a model and allow team members to modify parts of the model. If there are problems with modified model, a model versioning system can roll back the model into a previously versioned state. It can also be used to keep multiple generations of PSMs from an input PIM. OMG currently has the MOF 2.0 versioning and development lifecycle in a final

adopted specification. We believe that the model representation based on eMOF can implement versioning support and the model manipulation APIs can be refactored to support multi-version of models.

9.3.2 Additional Middleware Target for each Profile

Each profile from chapter 3 to 7 contains two implementations from 2 middleware libraries. However, to show that the framework is really extensible, we need to create and plugin new transformers to support more middleware targets. The Table 9 - 1 below shows examples of middleware targets. The ones in underlines have transformations.

Object Service	*Middleware Libraries*
Persistence	Hibernate, EJB CMP, JDO, JDBC/SQL, XML, Toplink, SDO
Transaction	Hiberate transaction, JTA, JDBC transaction, EJB CMT
Naming	JNDI, UUID, WS UDDI, URN
Distribution	RMI, CORBA IOR, SOAP/XML
Messaging	JMS, EJB MDB, ESB, MQSeries

Table 9 - 1 Middleware choices for each object service

9.3.3 Additional Profiles

The framework provides a standard way to define new profiles. This dissertation defines five common object services. However, it does not cover all aspects of enterprise software. Some profiles such as security, real time and workflow are important and contain details large enough to write another thesis. Many works have been done in these areas [LBD02] [OMG05]. We believe that the Mercator framework can be used as a tool to prototype these profiles and transformers associated with them.

9.3.4 Vertical Domain Specific Frameworks

This dissertation focuses on infrastructure "horizontal" service provided by middleware. UML has a profile extension mechanism to describe domain specific languages and we believe that the framework can be extended to support transformations of these languages. A research question can be whether we can apply this framework into software packages in various domains like accounting, billing, insurance, telecommunication systems? Can we create a component and mark it with «PayrollSystem» to indicate a payroll system and parameterize it in separate annotation files? What APIs are added to the component from this stereotype? What are other stereotyped components needed? Is it possible to define a «tax» with an attribute { policy= "single", "marriedWithDependents", "marriedWithoutDependents" } so that tax is calculated by employees' marital status among others? The actual calculation is generated by the «tax» stereotype transformers and may be different depending on tax policy used each year. We believe that using profiles, software development based on object oriented frameworks can be simplified because the customization of the framework can be put in separate annotation files and this makes it easier to change parameters. However, it may not be trivial for large frameworks.

9.3.5 Transformation Optimization

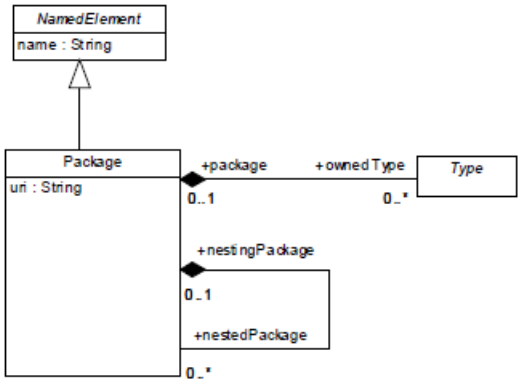
The prototype tool we developed took about 7 seconds to load, transform PIM-to-PSM and transform PSM-to-code for a model with 250 classes under a system with 1.83Ghz Intel Core Duo and 1.5GB RAM. From the result from the previous chapter, the transformation time scales almost linearly. This result is from a non-optimized transformation algorithm. However, we believe that we further improve the transformation performance by reorganizing model formats and AST data structure, preloading transformers' classes and regenerating only modified model elements. How much improvements can be made is an ongoing project we are pursuing.

9.3.6 IDE Integration

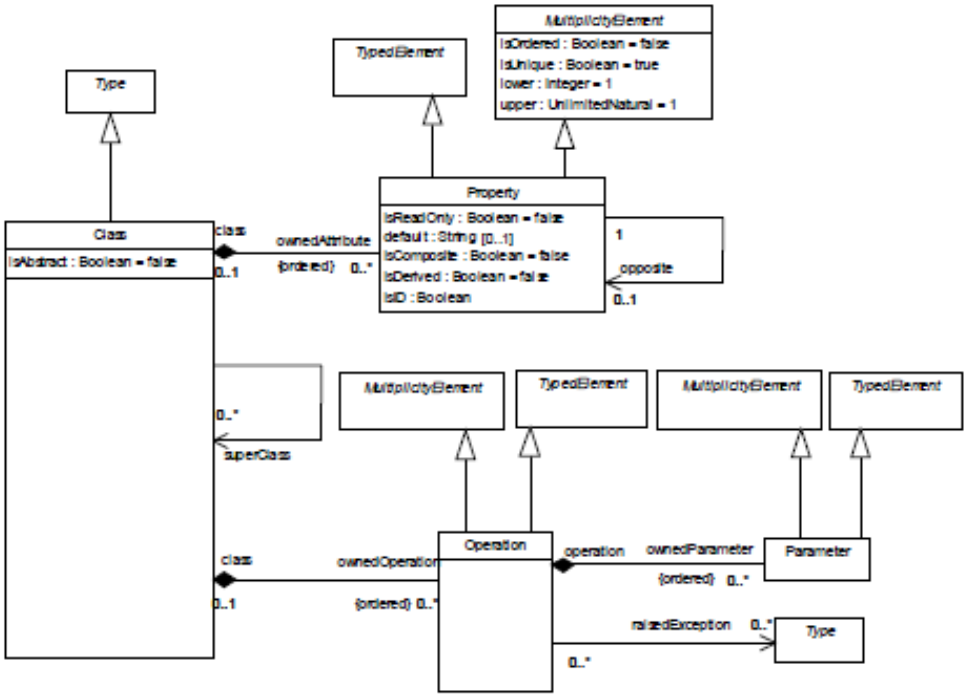
Developers usually use an IDE for their work and do not want to learn another tool. We believe that our tool will be more useful if it is integrated into existing IDEs. Not only does the integration provide single environment to manipulate, transform and view

models in different ways, but we can also use the IDEs' internal ASTs to represent software models. However, this may be a challenge if the IDEs' ASTs do not support extensions required by the framework. One possible solution is to create a bridge between the IDEs' ASTs and the Mercator AST. Mercator's specific changes in the source code AST will be stored in the Mercator AST while common structure will be stored in the IDE's AST.

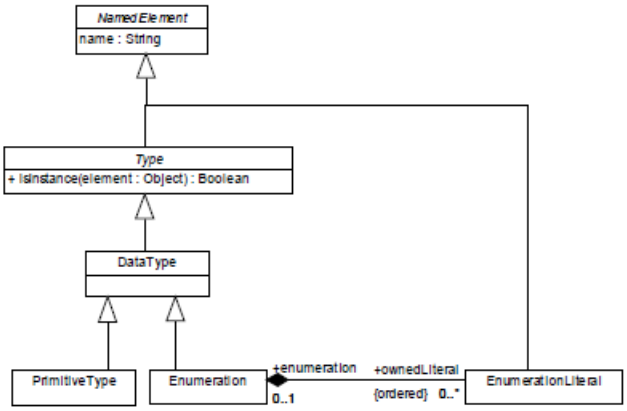
Appendix A: EMOF Model Elements



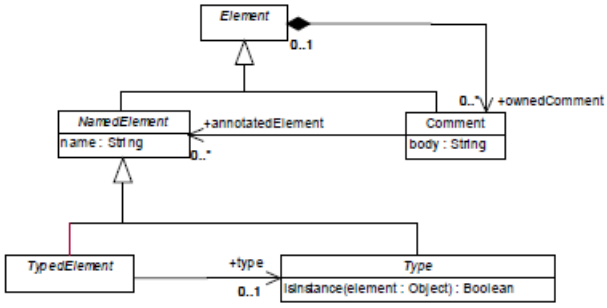
EMOF Packages



EMOF Classes



EMOF Datatypes



EMOF Types

Appendix B: IASTNode and AbstractFactory Interfaces

```
package org.mercator;

…

public interface IAstNode extends IVisitable {
    public RootNode getRootNode();
    public IAstNode getParentNode();
    public void setParentNode(IAstNode parent);

    public AbstractFactory getFactory();

    public String getMetaName();
    public String getQualifiedMetaName();
    public String getQualifiedName();

    public String getAttribute(String label);
    public String getAttribute(String label, String defaultValue);
    public void setAttribute(String label, String value);

    public String getLabel();

    public IAstNode getChild(int index);
    public int getChildCount();
    public boolean isLeaf();
    public Element getXmlElement();

    public String getIconFileName();

    public boolean containsExceptions();
    public void removeAllExceptions();
    public void addException(AbstractException anException);
    public void removeException(AbstractException anException);
    public List getExceptions();

    public void addStereotype(String qualifiedName);
    public boolean containsStereotype(String qualifiedName);
```

```
    public void removeStereotype(String qualifiedName);
    public void copyStereotypes(IAstNode pimRootNode);
    public void replaceStereotype(String original, String target);
    public List getAppliedStereotypeQualifiedNames();
    public String getAppliedStereotypesString();
    public boolean isStereotypeApplied(String qualifiedName);

    public Object getChild(Class ofClass);
    public List getChildren();
    public List getChildrenByXmlName(String xmlName);
    public List getChildren(Class ofClass);
    public String generateXmlString();
    public void removeChild(IAstNode astNode);
    public String getFileName();
    public String getNameDelimiter();

    public void setGenerated(boolean trueOrFalse);
    public boolean isGenerated();

    public IAstNode basicFindElement(String tagName, String elementName);
    public IAstNode findChild(String tagName, String attributeName);
    public IAstNode findChildDeep(String tagName, String attributeName);
    public IAstNode findType(String type);

    public String getBody();
    public void setBody(String newContents);

}
```

Listing B - 1 IAstNode interface

```
package org.mercator;
…
public abstract class AbstractFactory implements Observable {
    private String forNameSpace;
    private Set<Observer> observers = new HashSet<Observer>();

    private static final Log log = LogFactory.getLog(AbstractFactory.class);

    private Map<Element,IAstNode> cache = new HashMap<Element,IAstNode>();

    public IAstNode create(IAstNode parent, Element xmlElement) {
           IAstNode node = findNode(xmlElement);
```

```
            if (node == null) {
                  node = createNew(parent, xmlElement);
                  cache.put(xmlElement, node);
                  if (parent != null && xmlElement.getParent() == null) {
                        parent.getXmlElement().addContent(xmlElement);
                  }
            }
            return node;
      }

      public IAstNode findNode(Element xmlElement) {
            IAstNode node = (IAstNode) cache.get(xmlElement);
            return node;
      }

      public Element basicRead(String fileName) {
            log.info("Reading:" + fileName);
            Element root = JdomHelper.loadXMLElementFromFile(fileName);
            root.setAttribute("file", fileName);
            return root;
      }

      public IAstNode read(String fileName) {
            Element root = basicRead(fileName);
            IAstNode rootNode = create(null, root);
            return rootNode;
      }

      public void write(IAstNode root) {
                  String fileName =
root.getXmlElement().getAttributeValue("file");
                  if (Util.isNullOrEmpty(fileName)) {
                        log.warn("This AST does not have an associate file
name, save aborted.");
                        return;
                  }
                  log.info("Writing:" + fileName);
                  // do not save imported element
                  JdomHelper.storeXMLElementToFile(root.getXmlElement(),
fileName);
      }
```

```
    abstract public IAstNode createNew(IAstNode parent, Element xmlElement);
   …
    public String getForNameSpace() {
           return forNameSpace;
    }
    public void setForNameSpace(String forNameSpace) {
           this.forNameSpace = forNameSpace;
    }

    public void remove(IAstNode childNode) {
           cache.remove(childNode.getXmlElement());
    }

}
```

Listing B - 2 AbstractFactory interface

List of References

[Agr02] A. Agrawal, et al. "Generative Programming via Graph Transformations in the Model-Driven Architecture," In OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture. 2002.

[AMC03] D. Alur, D. Malks and J. Crupi. Core J2EE Patterns: Best Practices and Design Strategies, Second Edition, Prentice Hall, ISBN: 0131422464, 2003.

[Alm05] J. P. Almeida, R. Dijkman, M. Sinderen and L. F. Pires. "Platform-Independent Modelling in MDA: Supporting Abstract Platforms." MDAFA 2003/2004, LNCS 3599, 2005, pp. 174 – 188.

[Baa02] T. Baar. "Executable and Symbolic Conformance Tests for Implementation Models." Lecture Notes in Computer Science, Springer, ISSN: 0302-9743, volume 2426, 2002.

[Bez01] J. Bézivin. "From Object Composition to Model Transformation with the MDA," In Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), IEEE Computer Society, ISSN:1530-2067, 2001, p 350.

[BG04] C. Bauer and G. King. Hibernate In Action, Manning Publications, ISBN: 193239415X, 2004.

[BG81] P. Bernstein and N. Goodman. "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

[BGS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. "Overview of multidatabase transaction management," VLDB Journal, 1(2), 1992.

[BGD97] A. Behm, A. Geppert, K. R. Dittrich. "On the Migration of Relational Schemas and Data to Object-Oriented Database Systems," In Proc. 5th International Conference on Re-Technologies for Information Systems, Klagenfurt, Austria, 1997.

[BHG87] P. Bernstein, V. Hadzilacos and N. Goodman. Concurrency Control and Recovery in Database Systems, Addison-Wesley Publishing, 1987.

[Cap02] G. Caplat, J. Sourrouille. "Model Mapping in MDA," In Workshop in Software Model Engineering, Fifth International Conference on the Unified Modeling Language, 2002.

[Cat97] R. G. G. Cattell, et al. The object database standard: ODMG 2.0. Morgan Kaufmann Data Management Systems Series, ISBN: 1-55860-463-4, 1997.

[CE00] K. Czarnecki and U. W. Eisenecker. Generative programming: methods, tools, and applications.ACM Press/Addison-Wesley Publishing Co., 2000.

[CH03] K. Czarnecki and S. Helson. "Classification of Model Transformation Approaches," In OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[CN03] P. Clements and L. Northrop. Software Product Lines. Software Engineering Institute, Carnegie Mellon University, 2003.

[Cor04] The Object Management Group, Common Object Request Broker Architecture (CORBA/IIOP), formal/2004-03-01, http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[CR05] W. Cook and C. Rosenberger. "Native Query for Persistent Objects – A Design White Paper." Department of Computer Sciences, the University of Texas at Austin, http://www.cs.utexas.edu/~wcook/papers/NativeQueries/NativeQueries8-23-05.pdf, August 23, 2005.

[Cza03] K. Czarnecki and S. Helson. "Classification of Model Transformation Approaches," In OOPSLA 2003 Workshop in Generative Techniques in the context of Model Driven Architecture, 2003.

[ECA04] The Object Management Group, UML Profile for Enterprise Collaboration Architecture Specification. http://www.omg.org/cgi-bin/doc?formal/2004-02-05, 2004.

[Edoc04] The Object Management Group. Metamodel and UML Profile for Java and EJB, v1.0. http://www.omg.org/cgi-bin/doc?formal/2004-02-02, 2004.

[Eme05] M. J. Emerson. GME-MOF: An MDA Metamodeling Environment for GME, Master's Thesis, Vanderbilt University, EECS, May 2005.

[EMF] Eclipse Foundation. http://www.eclipse.org/emf/.

[Eng02] V. Englebert. "The Synchronization of Independent and Specific Models," In the 1st Workshop in Software Model Engineering at The Fifth International Conference on the Unified Modeling Language, Dresden, Germany, 2002.

[EP99] H. E. Eriksson, M. Penker. Business Modeling with UML - Business Patterns and Business Objects. John Wiley & Sons, ISBN: 0471295515, 1999.

[EUML2] Eclipse UML2 Project. http://www.eclipse.org/uml2/.

[Eva03] E. Evans. Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley. ISBN: 0321125215, 2003.

[Fow03] M. Fowler. Patterns of Enterprise Application Architecture. Addison Wesley Publishing, ISBN: 0321127420, 2003.

[Fra03] D. S. Frankel. Model Driven Architecture: Applying MDA to Enterprise Computing. OMG Press. ISBN: 0-471-31920-1, 2003.

[Fre05] French National Institute for Research in Computer Science and Control (INRIA). Model Transformation Language (MTL). http://model-ware.inria.fr/, January 2005.

[GGKH03] T. Gardner, C. Griffin, J. Koehler and R Hauser. "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard," In MetaModelling for MDA Workshop, 2003.

[GH02] G. Graw and P. Herrmann. "Verification of xUML Specifications in the context of MDA," In Workshop in Software Model Engineering, Fifth International Conference on the Unified Modeling Language, 2002.

[GJSB05] J. Gosling, B. Joy, G. Steele and G. Bracha. The Java Language Specification, Addison-Wesley Professional, 3rd edition, ISBN: 0321246780, 2005.

[GMP02] M. Gallardo, P. Merino and E. Pimentel. "Debugging UML designs with model checking." Journal of Object Technology, ISSN: 1660-1769, volume 1 issue 2, 2002, p101.

[Gog02] M. Gogolla, A. Lindo, M. Richters, P. Ziemann. "Metamodel Transformation of Data Models," In Workshop in Software Model Engineering, Fifth International Conference on the Unified Modeling Language, 2002.

[GSCK04] J. Greenfield, K. Short, S. Cook and S. Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. ISBN: 0471202843, Wiley, 2004.

[GS87] H. Garcia-Molina and K. Salem, "Sagas," Proceedings of ACM SIGMOD, San Francisco, CA, 1987, pp. 249–259.

[GTT02] S. Gerard, F. Terrier, Y. Tanguy. “Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML.” Lecture Notes in Computer Science, ISSU 2426, ISSN: 0302-9743, 2002, pp 260-269.

[Hib06] Hibernate relational persistence for Java and .NET, JBoss Inc., http://hibernate.org.

[ITU05] International Telecommunication Union. X.500 : Information technology - Open Systems Interconnection - The Directory: Overview of concepts, models and services. http://www.itu.int/rec/T-REC-X.500/e.

[JDO04] Java Data Objects Specification. Java Community Process. http://www.jcp.org/en/jsr/detail?id=12.

[Jpe02] iBATIS JPetStore project. http://sourceforge.net/projects/ibatisjpetstore/, 2002.

[Kob04] C. Kobryn. “UML 3.0 and the future of modeling.” Software and Systems Modeling, Springer Verlag, New York, 2004.

[Kov02] J. Kovse, T. Härder. “Generic XMI-Based UML Model Transformations,” In Proc. 8th Int. Conf. on Object-Oriented Information Systems (OOIS'02), 2002.

[KVR02] V. Kulkarni, R. Venkatesh, S. Reddy. “Generating Enterprise Applications from Models,” In OOIS-MDSD Workshop 2002.

[Lam86] B. Lampson. “Designing a global name service,” In Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, 1986, pp 1-10.

[LBD02] T. Lodderstedt, D. Basin, J. Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security.” Lecture Notes in Computer Science. Springer Berlin / Heidelberg. ISSN: 0302-9743. Volume 2460, 2002, pp 426-441.

[Linq06] Microsoft Corporation. LINQ Project. http://msdn.microsoft.com/data/ref/linq/.

[MBB06] E. Meijer, B. Beckman, G. Bierman. "LINQ: reconciling object, relations and XML in the .NET framework," In Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ISBN: 1-59593-434-0, 2006.

[MDA01] The Object Management Group. MDA Specifications webpage. http://www.omg.org/mda/specs.htm.

[MDR] Netbeans.org's Metadata Repository (MDR) Project. http://mdr.netbeans.org/.

[MN82] D. A. Menasce and T. Nakanishi, "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," Information Systems 7, 1, 1982.

[Mig02] M. Miguel, et al. "Specifications of Model Transformations Based on Meta-Templates," In Workshop in Software Model Engineering, Fifth International Conference on the Unified Modeling Language, 2002.

[Mof03] The Object Management Group. MOF 2.0 Core Final Adopted Specification. ptc/03-10-04, 2003.

[Mos81] J. Eliot B. Moss, Nested transactions: An Approach to Reliable Distributed Computing, Ph.D. thesis, MIT/LCS/TR-260, Massachusetts Institute of Technology, 1981.

[Obj05] The ObjectWeb consortium. http://middleware.objectweb.org/.

[OMG05] The Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. formal/05-01-02. http://www.omg.org/docs/formal/05-01-02.pdf, 2005.

[QVT05] The Object Management Group. MOF Query/Views/Transformations, ptc/05-11-01, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, 2005.

[Ray95] K. Raymond, Reference Model for Open Distributed Processing (RM-ODP) Introduction. 1995.

[RFBO01] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe. "The architecture of a UML virtual machine," In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. ACM Press, ISBN: 1-58113-335-9, 2001, pp 327-341.

[RM-ODP] ISO/IEC IS 10746. Reference Model for Open Distributed Processing. http://www.rm-odp.net/publications.html.

[Sch02] D. Schmidt, et al. "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications," In OOPSLA 2002 Workshop in Generative Techniques in the context of Model Driven Architecture. 2002.

[Sen03] S. Sendall, “Combining Generative and Graph Transformation Techniques for Model Transformation: An Effective Alliance?” Submitted to the OOPSLA 2003 Generative techniques in the conext of MDA Workshop.

[SFS04] R. Silaghi, F. Fondement and A. Strohmeier. "Towards an MDA-Oriented UML Profile for Distribution," In Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC, 2004.

[Sil03] R. Silaghi. “Better Generative Programming with Generic Aspects,” In OOPSLA 2003 Workshop in Generative Techniques in the context of Model Driven Architecture, 2003.

[Sil06] R. Silaghi. “Model-Driven Engineering of Middleware-Mediated Distributed Systems.” Swiss Federal Institute of Technology in Lausanne. PhD. Thesis. http://lgl.epfl.ch/members/silaghi/. To be published in 2006.

[Sim96] C. Simonyi. “Intentional programming: Innovation in the legacy age.” IFIP Working group 2.1. http://www.research.microsoft.com/research/ip/, 1996.

[Sin96] P. Sinha. Distributed Operating Systems: Concepts and Design. Wiley-IEEE Press, ISBN: 0780311191, 1996.

[Sun06] Sun Microsystems, Java Enterprise Edition APIs and Documentation, http://java.sun.com/javaee/reference/, 2006.

[SZ87] K. E. Smith, S. B. Zdonik. "Intermedia : a case study of the differences between relational and object-oriented database systems," In conference proceedings on Object-oriented programming systems, languages and applications, ISSN:0362-1340, 1987, pp 452-465.

[Tra05] L. Tratt. The MT model transformation language. Technical report TR-05-02, Department of Computer Science, King’s College London, 2005.

[Top06] Toplink, Oracle corporation. http://www.oracle.com/technology/products/ias/toplink/index.html.

[UML05] The Object Management Group. Unified Modeling Language 2.0: Superstructure. formal/05-07-04. http://www.omg.org/cgi-bin/doc?formal/05-07-04, 2005.

[UmlCor02] The Object Management Group. UML Profile for CORBA v1.0, formal/02-04-01, http://www.omg.org/technology/documents/formal/profile_corba.htm, 2002.

[UmlEjb01] The Java Community Process. Java Specification Request 26: UML/EJB Mapping Specification, http://jcp.org/en/jsr/detail?id=26, 2001.

[VKZ04] M. Voelter, M. Kircher, U. Zdun. Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware, John Wiley & Sons, ISBN: 0470856629, 2004.

[WJ03] W. Witthawaskul and R. Johnson. "Specifying Persistence in Platform Independent Models," In the 2nd Workshop in Software Model Engineering at The Sixth International Conference on the Unified Modeling Language, UML 2003, San Francisco, California, USA, 2003.

[WJ04] W. Witthawaskul and R. Johnson. "An Object Oriented Model Transformer Framework based on Stereotypes," In the 3rd Workshop in Software Model Engineering at The Sixth International Conference on the Unified Modeling Language, UML 2004, Lisbon, Portugal, 2004.

[WJ05] W. Witthawaskul and R. Johnson. "Transaction Support Using Unit of Work Modeling in the Context of MDA," edoc, pp. 131-141, Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05), September 19-23, 2005.

[Xsl99] The World Wide Web Consortium (W3C). XSL Transformations Version 1.0, http://www.w3.org/TR/xslt, 1999.

[Zia02] T. Ziadi, B. Traverson, J. Jezequel, “From a UML Platform Independent Component Model to Platform Specific Component Models,” In Workshop in Software Model Engineering, Fifth International Conference on the Unified Modeling Language, 2002.

Author's Biography

Weerasak Witthawaskul was born in Bangkok, Thailand, on November 27, 1970. He graduated first class honors from the King Mongkut's Institute of Technology at Lakrabang in 1991 with a bachelor degree in computer engineering. For several years he worked in a computer industry and was recognized for his work such as an IBM Thailand business contribution award in 1993 and three NCR CPC awards in 1994-1996. He received an MBA degree from Thammasat University in 1996 before he came abroad to study a graduate degree in computer science at the University of Illinois at Urbana-Champaign. During his first year of study, he won an AT&T Asia Pacific Leadership Award and had held research assistantship positions at the university ever since. He finished his master degree in 2001 and served as a conference chair for the Pattern Languages of Programs in 2002. Following the completion of his Ph.D. in 2006, he would like to solve challenging software engineering problems in company research labs or manage large scale software development in large organizations.